

Oracle® Database

Application Developer's Guide - Rules Manager and Expression
Filter

10g Release 2 (10.2)

B14288-01

June 2005

Copyright © 2003, 2005, Oracle. All rights reserved.

Primary Author: Aravind Yalamanchi and Rod Ward

Contributor: William Beauregard, Timothy Chorma, Lory Molesky, Dieter Gawlick, Helen Grembowicz, Deborah Owens, and Jagannathan Srinivasan

The Programs (which include both the software and documentation) contain proprietary information; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. This document is not warranted to be error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose.

If the Programs are delivered to the United States Government or anyone licensing or using the Programs on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software—Restricted Rights (June 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and we disclaim liability for any damages caused by such use of the Programs.

Oracle, JD Edwards, PeopleSoft, and Retek are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

Contents

Preface	xi
Audience	xi
Documentation Accessibility	xi
Related Documents	xii
Conventions	xii
What's New in Rules Manager and Expression Filter?	xiii
Oracle Database 10g Release 2 (10.2) New Features in Rules Manager and Expression Filter	xiii
1 Introduction	
1.1 What is a Rule?	1-2
1.2 Developing Rules Applications	1-4
Part I Rules Manager	
2 Rules Manager Concepts	
2.1 Rules Terminology	2-1
2.2 Database Representation of a Rule Class and Rules	2-3
2.3 Creating Rules Applications That Use Simple or Non-Composite Events	2-5
2.4 Creating Rules Applications That Use Composite Events	2-7
2.4.1 How to Create a Rules Application That Uses Composite Events	2-8
2.4.2 Evaluating Composite Events Using Complex Rule Conditions	2-11
2.5 Setting Event Management Policies (Rule Class Properties) for Rule Applications	2-11
2.6 Creating Rules Applications That Span Multiple Tiers	2-12
2.7 Using Rules Manager with SQL*Loader and Export/Import Utilities	2-12
2.7.1 SQL*Loader	2-12
2.7.2 Export/Import	2-13
3 Event Management Policies	
3.1 Consumption of Events	3-1
3.2 Conflict Resolution	3-3
3.3 Ordering of Rule Execution	3-4
3.4 Duration of Events	3-4
3.5 Equality	3-6

3.6	Storage Properties	3-7
3.7	AUTOCOMMIT	3-8
3.8	DML Events	3-9
3.9	Rule Class Property Dependencies and Defaults.....	3-9
3.10	Rules Specified on Relational Tables.....	3-10
3.11	Rules Conditions For XML Events	3-11
3.12	Rule Conditions with Spatial Predicates	3-12
3.13	Database State in Rule Conditions	3-13

4 Rule Conditions

4.1	Support for Incremental Evaluation of Rules	4-1
4.2	Rule Conditions with Sequencing	4-4
4.3	Rule Conditions with Negation	4-5
4.4	Rule Conditions with Set Semantics	4-8
4.5	Rule Conditions with Any n Semantics.....	4-9

5 Rules Applications That Span Multiple Tiers

5.1	Creating Rules Applications That Span Multiple Tiers.....	5-1
5.2	Modes of Operation.....	5-4
5.2.1	Single Tier Mode	5-4
5.2.2	Multitier Mode	5-4
5.2.2.1	Actions in the Mid-Tier.....	5-5

6 Rules Manager Object Types

7 DBMS_RLMGR Package

8 Rules Manager Views

8.1	USER_RLMGR_EVENT_STRUCTS View	8-1
8.2	USER_RLMGR_RULE_CLASSES View	8-1
8.3	USER_RLMGR_RULE_CLASS_STATUS View.....	8-2
8.4	USER_RLMGR_PRIVILEGES View	8-2
8.5	USER_RLMGR_COMPRCLS_PROPERTIES View	8-3

9 Rules Manager Use Cases

9.1	Law Enforcement Rules Application	9-1
9.2	Order Management Rules Application.....	9-8

Part II Expression Filter

10 Oracle Expression Filter Concepts

10.1	What Is Expression Filter?	10-1
10.1.1	Expression Filter Usage Scenarios.....	10-1
10.2	Introduction to Expressions	10-3
10.2.1	Defining Attribute Sets	10-4

10.2.2	Defining Expression Columns	10-7
10.2.3	Inserting, Updating, and Deleting Expressions	10-8
10.3	Applying the SQL EVALUATE Operator	10-9
10.4	Evaluation Semantics	10-11
10.5	Granting and Revoking Privileges	10-11
10.6	Error Messages	10-12
11	Indexing Expressions	
11.1	Concepts of Indexing Expressions.....	11-1
11.2	Indexable Predicates.....	11-1
11.3	Index Representation.....	11-2
11.4	Index Processing	11-3
11.5	Predicate Table Query	11-5
11.6	Index Creation and Tuning	11-5
11.7	Index Usage	11-7
11.8	Index Storage and Maintenance	11-8
12	Expressions with XPath Predicates	
12.1	Using XPath Predicates in Expressions	12-1
12.2	Indexing XPath Predicates.....	12-2
12.2.1	Indexable XPath Predicates	12-3
12.2.2	Index Representation	12-3
12.2.3	Index Processing	12-4
12.2.4	Index Tuning for XPath Predicates	12-5
13	Expressions with Spatial Predicates	
13.1	Using Spatial Predicates in Expressions	13-1
13.2	Indexing Spatial Predicates	13-3
14	Using Expression Filter with Utilities	
14.1	Bulk Loading of Expression Data	14-1
14.2	Exporting and Importing Tables, Users, and Databases.....	14-2
14.2.1	Exporting and Importing Tables Containing Expression Columns.....	14-2
14.2.2	Exporting a User Owning Attribute Sets	14-3
14.2.3	Exporting a Database Containing Attribute Sets	14-3
15	SQL Operators and Statements	
	EVALUATE	15-2
	ALTER INDEX REBUILD.....	15-4
	ALTER INDEX RENAME TO	15-5
	CREATE INDEX.....	15-6
	DROP INDEX	15-9

16 Object Types

17 Management Procedures Using the DBMS_EXPFIL Package

18 Expression Filter Views

18.1	USER_EXPFIL_ASET_FUNCTIONS View	18-1
18.2	USER_EXPFIL_ATTRIBUTES View	18-2
18.3	USER_EXPFIL_ATTRIBUTE_SETS View	18-2
18.4	USER_EXPFIL_DEF_INDEX_PARAMS View	18-2
18.5	USER_EXPFIL_EXPRESSION_SETS View	18-3
18.6	USER_EXPFIL_EXPRSET_STATS View	18-3
18.7	USER_EXPFIL_INDEX_PARAMS View	18-4
18.8	USER_EXPFIL_INDEXES View	18-4
18.9	USER_EXPFIL_PREDTAB_ATTRIBUTES View	18-5
18.10	USER_EXPFIL_PRIVILEGES View	18-5

A Managing Expressions Defined on One or More Database Tables

B Application Examples

C Internal Objects

C.1	Attribute Set or Event Structure Object Type	C-1
C.2	Expression Filter Internal Objects	C-2
C.2.1	Expression Validation Trigger	C-2
C.2.2	Expression Filter Index Objects	C-2
C.2.3	Expression Filter System Triggers	C-2

D Converting Rules Applications

D.1	Differences Between Expression Filter and Rules Manager	D-1
D.2	Converting an Expression Filter Application to a Rules Manager Application	D-3

E Installing Rules Manager and Expression Filter

F XML Schemas

G Implementing Various Forms of Rule Actions With the Action Callback Procedure

Index

List of Examples

10-1	Defining an Attribute Set From an Existing Object Type	10-5
10-2	Defining an Attribute Set Incrementally	10-5
10-3	Adding User-Defined Functions to an Attribute Set	10-6
10-4	Inserting an Expression into the Consumer Table	10-9
10-5	Inserting an Expression That References a User-Defined Function	10-9

List of Figures

1-1	Rules Manager Implementation Process for a Rules Application	1-3
2-1	Database Representation of Rule Class and Rules.....	2-4
4-1	Hierarchical View of the XML Tag Extensions.....	4-3
10-1	Expression Filter Implementation Process for a Rules Application	10-4
10-2	Expression Data Type.....	10-8
11-1	Conceptual Predicate Table	11-3
12-1	Conceptual Predicate Table with XPath Predicates	12-4

List of Tables

3-1	Valid and Invalid Rule Class Property Combinations	3-9
4-1	Relational View of the XML Tag Extensions.....	4-3
6-1	Rules Manager Object Types.....	6-1
7-1	DBMS_RLMGR Procedures	7-1
8-1	Rules Manager Views.....	8-1
8-2	USER_RLMGR_EVENT_STRUCTS View	8-1
8-3	USER_RLMGR_RULE_CLASS View	8-1
8-4	USER_RLMGR_RULE_CLASS_STATUS View.....	8-2
8-5	USER_RLMGR_PRIVILEGES View	8-2
8-6	USER_RLMGR_COMPRCLS_PROPERTIES View	8-3
15-1	Expression Filter Index Creation and Usage Statements	15-1
16-1	Expression Filter Object Types.....	16-1
17-1	DBMS_EXPFIL Procedures.....	17-1
18-1	Expression Filter Views.....	18-1
D-1	Implementation Differences Between Expression Filter and Rules Manager for Rules Applications That Use a Primitive (Simple) Event	D-2
G-1	TravelPromotion Rule Class Table.....	G-1
G-2	Modified TravelPromotion Rule Class Table	G-2

Preface

Oracle Database Application Developer's Guide - Rules Manager and Expression Filter provides usage and reference information about Rules Manager, a feature in the Oracle Database that offers interfaces to define, manage, and enforce complex rules in the database and Expression Filter, a feature of Oracle Database and component of Rules Manager that stores, indexes, and evaluates conditional expressions in relational tables.

Audience

Oracle Database Application Developer's Guide - Rules Manager and Expression Filter is intended for application developers and DBAs who perform the following tasks:

- Use Event-Condition-Action (ECA) rules to integrate processes and automate workflows
- Use the database to store and evaluate large sets of conditional expressions
- Use the database to define, manage, and enforce complex rules

This manual assumes a working knowledge of application programming and familiarity with SQL, PL/SQL, XML, and basic object-oriented programming to access information in relational database systems.

Documentation Accessibility

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

<http://www.oracle.com/accessibility/>

Accessibility of Code Examples in Documentation

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

Accessibility of Links to External Web Sites in Documentation

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

TTY Access to Oracle Support Services

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

Related Documents

For more information, see these Oracle resources:

- *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 - Fundamentals*
- *Oracle Database Application Developer's Guide - Object-Relational Features*
- *Oracle Database PL/SQL Packages and Types Reference*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, terms defined in text or the glossary, or important parts of an example.
<i>italic</i>	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
monospace	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

What's New in Rules Manager and Expression Filter?

This section describes new features of Oracle Database 10g Release 2 (10.2) and provides pointers to additional information. New features information from previous releases is also retained to help those users migrating to the current release.

The following section describes the new features in Rules Manager and Expression Filter:

- [Oracle Database 10g Release 2 \(10.2\) New Features in Rules Manager and Expression Filter](#)

Oracle Database 10g Release 2 (10.2) New Features in Rules Manager and Expression Filter

- Rules Manager

Rules Manager is a new feature for Oracle Database 10g Release 2 (10.2). Rules Manager uses the Expression Filter and Oracle Object Relational features to give Oracle database the features of a special purpose rules product, but with greater scalability and better operational characteristics.

See Also: [Part I, "Rules Manager"](#), which contains all the chapters that describe this feature

- Expression Filter — cost-based optimizer support for the `EVALUATE` operator

The selectivity and the cost of an Expression Filter index are computed when statistics are collected on the expression column, the index, or the table storing expressions. These statistics are stored in the Expression Filter dictionary and they are used to determine the optimal execution plan for the query with `EVALUATE` operator.

See Also: [Section 11.7, "Index Usage"](#)

- Expression Filter — attribute set with default values for some attributes

At the time of attribute set creation, default values can be assigned to one or more elementary attributes so that these values are available at the time of expression evaluation. Default values cannot be specified for table alias elementary attributes in the attribute set.

See Also: See the `ADD_ELEMENTARY_ATTRIBUTE` procedure in [Chapter 17, "Management Procedures Using the DBMS_EXPFIL Package"](#)

- Expression Filter — index support for XPath predicates with the Namespace specification

See Also: See the `EXF$XPATH_TAG` object type in [Chapter 16, "Object Types"](#)

Introduction

Application developers use rules to integrate business processes and automatically respond to events created by workflows. However, these rules are often embedded in code modules or a special purpose memory-based rules repository making maintenance of them challenging. Rules that are managed in Oracle Database keep pace with changing business conditions and are always up-to-date; rules are easily changed with SQL and are not hard-coded in your application or loaded into a memory-based rules repository. Rules can be evaluated efficiently with the complete business context stored in your Oracle Database as well as data provided by your application. Event response is flexible; rules can trigger actions in Oracle Database or in your application, or both.

Rules Manager application programming interface (APIs) define, manage, and enforce complex rules in the Oracle Database with better scalability and operational characteristics than a special purpose rules product. Additionally, Rules Manager as a database feature can be used in multiuser and multisession environments.

Rules Manager can model any event-condition-action (ECA)-based system ranging from the simplest single event-single rule system to rule-based systems that involve millions of events and millions of rules. Applications for Rules Manager include information distribution, task assignment, event-based computing, radio frequency ID (RFID), supply chain, enterprise application integration (EAI), business asset management (BAM), and business process management (BPM).

Rules Manager processes an event or a group of events for a set of rules that are based on ECA semantics. An event can be an individual entity (simple or primitive event) or a group of events (composite event). Rules Manager models complex event scenarios using SQL and XML based rule condition language. An event can be incoming application data or data stored as rows in one or more relational tables. Rules Manager supports the Oracle-supplied XMLType data type, which allows it to process XML events.

When an event happens, and if a rule condition evaluates to true for that event, then a prescribed rule action is performed, which can be either executed immediately or obtained as a list of rules that evaluate to true for the event for later execution by the application or some other component and that can be queried.

While processing a set of rules for an event, Rules Manager enforces various event management policies, including conflict resolution among composite events or groups of matching rules, ordering of events, lifetime of an event, and sharing events across multiple rule executions.

The concept of rules is briefly introduced in [Section 1.1](#) followed by an overview of Rules Manager features. [Section 1.2](#) describes some general concepts about developing rules applications using Rules Manager.

If you have an existing Expression Filter application and want to upgrade it to a Rules Manager application, first see [Section D.1](#), which describes an implementation of Expression Filter and Rules Manager. Next, see [Section D.2](#), which describes the process of upgrading an Expression Filter application to a Rules Manager application.

1.1 What is a Rule?

A rule is a directive to guide or influence a process behavior. A rule consists of a conditional expression that is specified using the attributes defined in a corresponding event structure and a rule action that takes place when the rule condition is satisfied by an instance of the event structure. Event management policies define how an event instance is handled once the rule action is executed. This, in a nutshell, describes how a typical rules-based system works.

Typically, rules follow **Event-Condition-Action (ECA) rule** semantics where an event happens and if a rule condition evaluates to true for this event, then some prescribed action is performed. The ECA components are defined as:

- **Event** -- the state information for the process
- **Condition** -- the Boolean condition that evaluates to true or false for the event
- **Action** -- the action to be carried out if the rule condition evaluates to true for the event

The standard notation for ECA rules is:

```
ON    <event structure>
IF    <condition>
THEN  <action>
```

where, the ON clause identifies the event structure for which the rule is defined, the IF clause specifies the rule condition, and the THEN clause specifies the rule action.

An example of a rule is the following: If a customer chose to fly Abcair Airlines to Orlando and if his stay in Orlando is more than 7 days, then offer an Acar rental car promotion to him. Using the ECA notation, this rule is:

```
ON
    AddFlight (Custid, Airline, FromCity, ToCity, Depart, Return)
IF
    Airline = 'Abcair' and ToCity = 'Orlando' and Return-Depart >= 7
THEN
    OfferPromotion (CustId, 'RenralCar', 'Acar')
```

where:

The ON structure identifies the event structure for this rule.

The IF clause defines the rule condition using variables in the event structure.

The THEN clause defines the commands that represent the action for this rule.

Rules Manager

Rules Manager, a feature of Oracle Database, offers interfaces to define, manage, and enforce complex rules in the database. The five elements of a Rules Manager application are:

1. An event structure that is defined as an object type with attributes that describe specific features of an event.
2. A rule consisting of a condition and action preferences.

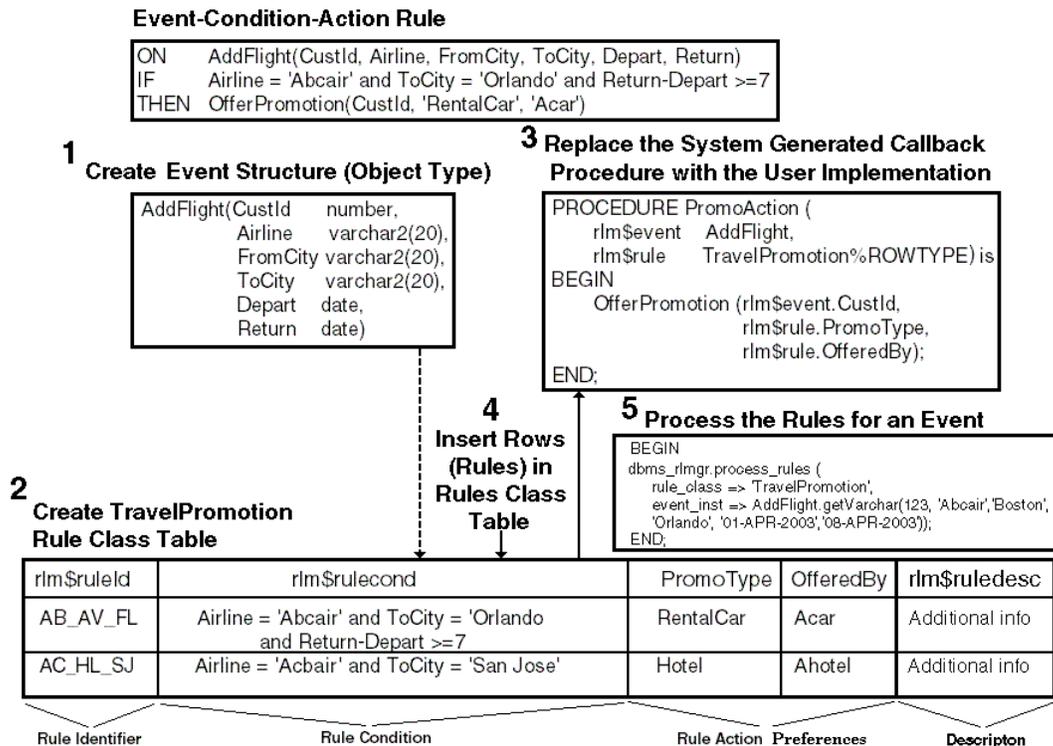
- A rule condition is expressed using the attributes defined in the event structure.
 - Rule action preferences determine the exact action for each rule and specify the details for the action.
3. A rule class that is a database table that stores and groups the rules defined for an event structure.
 4. An action callback PL/SQL procedure that implements the rule actions for the rule class. The implementation can rely on some attributes of the event structure and the action preference associated with the rules.
 5. A results view that configures a rule class for external rule action execution.

Rules Manager supports XML-based condition language, SQL commands for rule specification, automated tracking of events, declarative management of event policies, rule actions, and an application programmatic interface (API).

Rules Manager supports primitive (simple) and composite events. Rules Manager is appropriate for any rules-based applications requiring composite events. Rules Manager supports complex rule conditions involving negation, set semantics, Any n construct, and sequencing. Rules Manager supports incremental evaluation of rules involving composite events. Complex rule conditions are specified using XML tags within conditional expressions in the SQL WHERE clause format. Rule class event management policies such as consumption, conflict resolution, and duration can be enforced for each rule application. Figure 1–1 shows the process steps for creating and implementing a Rules Manager rules application. Section 2.3 describes these steps in more detail.

For more information about creating, using, and maintaining Rules Manager applications, see Part I, "Rules Manager", Chapter 2 through Chapter 9.

Figure 1–1 Rules Manager Implementation Process for a Rules Application



1.2 Developing Rules Applications

Developing a rules application using Rules Manager requires a somewhat different approach toward application development. Typically, you would scan through a new API and other reference material, then create some sample scripts based on the examples to get a feel for how the feature works. Next, you might begin to apply these methods learned to your own application. However, this is where you might get bogged down in the detail of the implementation because the approach to Rules Manager application development uses a somewhat different focus. The focus is on the decision points that already exist in your application and that is all. You need not focus on all the supporting parts of your application that do not necessarily pertain to these decision points.

As an application developer you must ask yourself the following questions:

- Where are the decision points in my application?
- What are the decisions that each decision point is making?
- How is each decision being made?
- Once a decision is made how is it executed in the application?

Note that each decision point may use one or more rules that may involve one or more events happening in some sequence.

Once you determine the decision points in your application, you integrate the Rules Manager into your application by using the standard notation for ECA rules to model each decision point as described in [Section 1.1](#). It is best to keep your approach as simple as possible.

For example, using Rules Manager for the simplest case, if your application has a decision point that uses one or more rules each relying on a single instance of an event structure that happens in the application, you would define a primitive event structure to model this event activity. In a complex event scenario, if your application has another decision point that uses one or more rules, each relying on multiple instances of the same or different event structures that happen in some sequence, define a composite event structure consisting of separately defined primitive event structures for each individual event that happens. The composite event structure couples these primitive events together to model the composite event activity. Next, create the rule class. Creating the rule class implicitly creates the rule class table containing an expression column to store the rule conditions and one or more action preferences columns that are used to determine the appropriate action when the rule evaluates to true. In addition to the rule class table, the previous step also creates an action callback procedure that you can modify to execute the action for the matching rules.

This unique approach lets you quickly integrate Rules Manager into existing applications as easily as if it were a new application because you only need to focus on the decision points contained in your application or in your data analysis for a new application. Remember Rules Manager stores, processes, and matches rules with instances of either incoming single events or groups of events to resolve the rules concentrated around each decision point. The object then becomes how best to model these decision points using Rules Manager. This is explained in [Part I, "Rules Manager"](#).

Part I

Rules Manager

This part introduces developing applications using Rules Manager feature.

Part I contains the following chapters:

- [Chapter 2, "Rules Manager Concepts"](#)
- [Chapter 3, "Event Management Policies"](#)
- [Chapter 4, "Rule Conditions"](#)
- [Chapter 5, "Rules Applications That Span Multiple Tiers"](#)
- [Chapter 6, "Rules Manager Object Types"](#)
- [Chapter 7, "DBMS_RLMGR Package"](#)
- [Chapter 8, "Rules Manager Views"](#)
- [Chapter 9, "Rules Manager Use Cases"](#)

Rules Manager Concepts

Rules Manager is a feature of Oracle Database that uses the Expression Filter and object relational features to provide the features of a special-purpose rules engine with greater scalability and better operational characteristics.

2.1 Rules Terminology

Rules Manager uses the following terminology:

- An **event structure** is an object (abstract) type that is defined with a set of attributes that describes the specific features of an event. For example, it is the data structure that captures the customer flight information, using variables, such as Airline, Customer Id, From City, and so forth. The object type definition of the AddFlight event structure is as follows:

```
TYPE AddFlight AS OBJECT (
  CustId      NUMBER,
  Airline     VARCHAR2(20),
  FromCity    VARCHAR2(30),
  ToCity      VARCHAR2(30),
  Depart      DATE,
  Return      DATE);
```

- An event is the instantiation of the event structure, so each instance of the event structure is an event. For example, these are three events:

```
AddFlight (123, 'Abcair', 'Boston', 'Orlando', '01-Apr-2003', '08-Apr-2003');
AddFlight (234, 'Acabair', 'Chicago', 'San Jose', '01-Aug-2003',
           '10-Aug-2003');
AddFlight (345, 'Acabair', 'New York', 'San Jose', '22-Jun-2003',
           '24-Jun-2003');
```

- Events are classified into two types:
 - **Primitive event** - represents an event that is assumed to be instantaneous and atomic in an application. A primitive event cannot be further broken down into other events and it either occurs completely or not at all. Each primitive event is typically bound to a specific point in time and the rules defined for the corresponding event structure can be fully evaluated with the event. For example, the AddFlight event is an example of a primitive event:

```
AddFlight (CustId, Airline, FromCity, ToCity, Depart, Return)
```

- **Composite event** - represents the combination of two or more primitive events. All primitive events included in the composite event can be bound to a time window and thus generated at different points in time. So the rules

defined for the composite event structure cannot be fully evaluated until all the corresponding primitive events are generated. For example, adding a second primitive event `AddRentalCar` to the `AddFlight` primitive event creates a composite event:

```
AddFlight (CustId, Airline, FromCity, ToCity, Depart, Return)
AddRentalCar (CustId, CarType, Checkout, Checkin, Options)
```

Because evaluation of rules for composite event structures must be deferred until all parts of a composite event are available, Rules Manager provides several ways of efficiently evaluating composite events.

See [Section 2.4](#) for more information about composite events and complex rule applications.

- A **rule class** is a database table that stores and groups a set of rules that share a common event structure. For example, this rule class of three rules is for the `AddFlight` event structure:

```
ON AddFlight (CustId, Airline, FromCity, ToCity, Depart, Return)
IF Airline = 'Abcair', and ToCity = 'Orlando'
THEN OfferPromtion (CustId, 'RentalCar', 'Acar')
```

```
ON AddFlight (CustId, Airline, FromCity, ToCity, Depart, Return)
IF Airline = 'Acbaair', and ToCity = 'Houston'
THEN OfferPromtion (CustId, 'RentalCar', 'Bcar')
```

```
ON AddFlight (CustId, Airline, FromCity, ToCity, Depart, Return)
IF ToCity = 'Orlando' and Return-Depart >7
THEN OfferPromtion (CustId, 'ThemePark', 'Ocean World')
```

- Rules are evaluated for an instance of the corresponding event structure. For example, the following event is used to evaluate the rules defined using the `AddFlight` event structure:

```
AddFlight (123, 'Abcair', 'Boston', 'Orlando', '01-Apr-2003', '08-Apr-2003');
```

- A rule is a row in a rule class table that has elements consisting of:
 - The **rule condition**, which is a conditional expression that is formed using the attributes defined in the event structure. For example, this is a rule condition using the attributes: `Airline`, `ToCity`, `Return`, and `Depart`:


```
Airline = 'Abcair' and ToCity = 'Orlando' and Return-Depart >= 7
```
 - The **rule action preferences**, which determine the exact action for each rule and specify the details for the action.

Typically, the actions associated with rules in the rule class are homogenous. For example, if a rule class is used to determine the discount offered during a checkout process, each rule in the class is associated with a specific discount percentage. For rules that match an event instance, these values are used to determine the appropriate action for the rule.

Action preferences can come in different forms, such as:

- * A list of literals that are bound as arguments to the common procedure, such as:

```
'RentalCar', 'Acar', 'Bcar',...
```

- * Dynamic PL/SQL commands, such as:

```
BEGIN OfferRentalPromotion(:1,'Acar'); END;
```

- An **action callback procedure** is a procedure that acts as an entry point for executing actions for all the rules in a rule class. This procedure is implemented to execute the action for each rule in the rule class based on the action preferences associated with the rule and the event attributes. For the previous example, the action callback procedure can be implemented to invoke the OfferPromotion procedure with the appropriate arguments.
- A **results view** configures a rule class for external action execution when the actions for each matching rule cannot be executed by means of an action callback procedure, such as applications that span multiple tiers.

The rules matching an event are available by querying this preconfigured view and the corresponding actions can be executed by the component issuing the query. This is useful when the action for certain rules is implemented in the application on multiple tiers. See [Section 2.6](#) for more information.

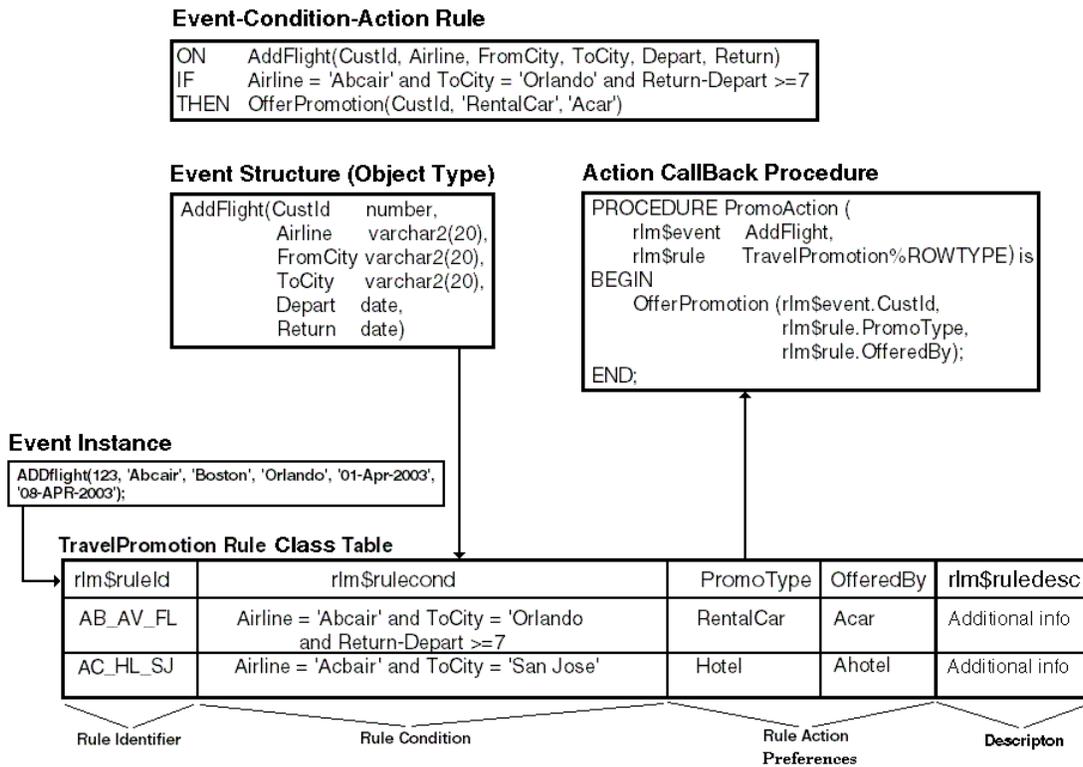
- The results from a rule evaluation are available through the results view until the end of the **rule session**. By default, the database session (from connect to disconnect) is considered the rule session. Alternatively, the reset session procedure (`dbms_rlmgr.reset_session()`) can be used to end a rule session and start a new session within a database session. Note that at the beginning of a rule session, the results view is empty.
- Rule class properties define the **event management policies** that Rules Manager enforces for each rules application. Two main policies discussed in this chapter are consumption and conflict resolution. **Consumption** refers to whether an event can be used for multiple rule executions or for just a single rule execution (see [Section 3.1](#)). **Conflict resolution**, or ordering, determines the order in which matching rules with various events are to be executed (see [Section 3.2](#)). [Section 2.5](#) and [Chapter 3](#) describe the complete set of event management policies that Rules Manager supports.

2.2 Database Representation of a Rule Class and Rules

Rules Manager uses a relational table to hold the contents of a rule class with each row in the table representing a rule. The rule class table minimally has three columns, one for rule identifiers (`rlm$ruleid`), one for rule conditions (`rlm$rulecond`), and one for the description of the rule (`rlm$ruledesc`). In addition, the rule class table can have one or more columns to store rule action preferences.

[Figure 2–1](#) shows a database representation of the TravelPromotion rule class and its rules for processing the AddFlight event instances.

Figure 2-1 Database Representation of Rule Class and Rules



The TravelPromotion rule class consists of the following columns:

- `rlm$ruleid` -- contains the unique rule identifier that identifies each rule within a rule class.
- `rlm$rulecond` -- contains the rule condition describing each rule; in this case, the rule condition, when satisfied, allows the promotion specified to be offered.
- `PromoType` -- contains one action preference that is used when the rule condition is satisfied, and in each case, the action callback procedure is called that executes the actions for the rules in the rule class; in this case, the type of promotion to be offered, such as a car rental promotion or hotel stay promotion is stored in this column. This value is used by the PromoAction action callback procedure to invoke the OfferPromotion procedure with the appropriate arguments.
- `OfferedBy` -- contains another action preference that is associated with the previous action preference column; in this case, it contains the name of the company offering the promotion.
- `rlm$ruledesc` -- contains a description of the rule in plain text provided by the person defining the rule.

An ECA rule is stored in a row of the TravelPromotion rule class table. The event structure, defined as an object type in the database, is associated with the rule condition column and this provides the necessary vocabulary for the rule conditions (stored in the column). The event structure, the rule class table, and the action callback procedure are all created as part of rule class creation.

Once all the rules are added to the rule class, events are ready to be processed and rules evaluated. At runtime, each rule in the rule class is processed against each instance of the event structure. When a rule evaluates to true for a particular event, the

PromoAction action callback procedure calls the designated OfferPromotion procedure using rule action preferences to execute the prescribed action of offering a specific type of promotion from a particular vendor. Rules Manager enforces various event management policies, such as conflict resolution when an event matches more than one rule, or immediate event consumption when the first match is found and no further evaluation is necessary. These and other event management policies are described in more detail in [Chapter 3](#).

[Section 2.3](#), [Section 2.6](#), and [Section 2.4](#) describe the process of creating rules applications that use a simple event, that span multiple tiers, and that use composite events, respectively. Though the basic five steps are the same for all three cases, the details vary, and some additional steps are necessary for multiple tier applications.

2.3 Creating Rules Applications That Use Simple or Non-Composite Events

The basic steps to create a rules application that uses a simple or non-composite event are as follows:

1. Create the event structure as an object type in the database.

Using the AddFlight example, the event structure is defined as:

```
CREATE TYPE AddFlight AS OBJECT (
  CustId      NUMBER,
  Airline     VARCHAR2(20),
  FromCity    VARCHAR2(30),
  ToCity      VARCHAR2(30),
  Depart      DATE,
  Return      DATE);
```

2. Create the rule class for the event structure.

Note: For successful creation of a rule class, you should have sufficient privileges to create views, object types, tables, packages, and procedures.

For this example, create the TravelPromotion rule class for the AddFlight event structure and define the PromoType and OfferedBy columns as its action preferences. This procedure takes the name of the rule class, the name of the existing event structure created in Step 1, the name of the action callback procedure, and the action preference specification as arguments. The action preferences specification defines the data types of action preferences that are associated with each rule in the rule class.

```
BEGIN
dbms_rlmgr.create_rule_class (
  rule_class   => 'TravelPromotion',
  event_struct => 'AddFlight',
  action_cbk   => 'PromoAction',
  actprf_spec => 'PromoType VARCHAR2(20),
                OfferedBy VARCHAR2(20)');
END;
```

Rule class creation creates a table to store the corresponding rule definitions and action preferences. The rule class table uses the same name as the rule class and it is created in the user's schema. The rule class table defines three columns to store

the rule identifiers, rule descriptions, and the rule conditions. In this example, the table also creates the rule action preferences columns specified with the previous command to store the action preferences.

```
TABLE TravelPromotion (
    rlm$ruleid      VARCHAR2(100),
    rlm$rulecond    VARCHAR2(4000),
    rlm$ruledesc    VARCHAR2(1000),
    PromoType       VARCHAR2(20),
    OfferedBy       VARCHAR2(20));
```

You can query the table to see the rules defined in the rule class as well as perform SQL INSERT, UPDATE, and DELETE operations to add, update, and delete rules.

Rule class creation implicitly creates the skeleton for a callback procedure to perform the action. The action callback procedure acts as an entry point for executing actions for all the rules in the rule class. The action callback is called once for every rule that matches an event. The implementation of the action callback procedure can rely on values in the event instance and the action preferences associated with the matching rule.

```
PROCEDURE PromoAction (rlm$event      AddFlight,
                       rlm$rule       TravelPromotion%ROWTYPE) is
BEGIN
    null;
    --- The action for the matching rules can be performed here.
    --- The appropriate action can be determined from the event
    --- instance and the action preferences associated with each rule.
END;
```

The action callback procedure, in this case, is created with the name the user provides and has two arguments:

- The event as an instance of the corresponding object type.
 - The action preferences as a ROWTYPE of the corresponding rule class table. The %ROWTYPE attribute provides a record type that represents a row in a table.
3. Replace the system-generated callback procedure with the user implementation to perform the appropriate action for each matching rule. The following action callback procedure can be implemented to invoke the OfferPromotion procedure with arguments obtained from the event instance and the rule definition:

For this example,

```
PROCEDURE PromoAction (
    rlm$event      AddFlight,
    rlm$rule       TravelPromotion%ROWTYPE) is
BEGIN
    OfferPromotion (rlm$event.CustId,
                   rlm$rule.PromoType,
                   rlm$rule.OfferedBy);
END;
```

In this example, the procedure OfferPromotion performs the action and each matching rule provides the appropriate action preferences. [Appendix G](#) shows alternate ways for implementing the action callback procedure for a different choice of action preferences.

4. Add rules to the rule class.

Adding rules consists of using the SQL `INSERT` statement to add a row for each rule. Each row inserted typically contains a rule identifier, a condition, and values for action preferences. The following rule is inserted into the `TravelPromotion` table:

```
INSERT INTO TravelPromotion (rlm$ruleid, PromoType, OfferedBy, rlm$rulecond)
VALUES
('UN_AV_FL', 'Rental Car', 'Acar',
'Airline= 'Abcair' and ToCity = 'Orlando' and Return-Depart >= 7');
```

5. Process the rules for an event.

Use the `dbms_rlmgr.process_rules()` procedure to process the rules in a rule class for an event instance. Processing the rules consists of passing in an event instance as a string of name-value pairs (generated using the `getVarchar()` procedure) or as an `AnyData` instance for an event consisting of binary data types as described in [Section 10.3](#). Recall that the Oracle supplied `getVarchar()` method is used to represent the data item as string-formatted name-value pairs when this is possible and that `AnyData` is an Oracle supplied object type that can hold instances of any Oracle data type, both Oracle supplied and user-defined.

The following example processes the rules in the `TravelPromotion` rule class for an `AddFlight` event instance using the `getVarchar()` function.

```
BEGIN
dbms_rlmgr.process_rules (
  rule_class => 'TravelPromotion',
  event_inst => AddFlight.getVarchar(987, 'Abcair', 'Boston', 'Orlando',
'01-APR-2003', '08-APR-2003'));
END;
```

The following example processes the rules in the `TravelPromotion` rule class for an `AddFlight` event instance using the `AnyData.ConvertObject()` procedure.

```
BEGIN
dbms_rlmgr.process_rules (
  rule_class => 'TravelPromotion',
  event_inst => AnyData.convertObject(AddFlight(987, 'Abcair', 'Boston',
'Orlando', '01-APR-2003', '08-APR-2003'));
END;
```

The previous command processes the rules in the `TravelPromotion` rule class for an `AddFlight` event instance and performs the action associated with each matching rule through the action callback procedure.

2.4 Creating Rules Applications That Use Composite Events

Probably the more common types of rules applications are those that use a composite event structure that combines two or more primitive events. Evaluating rule classes for composite events creates additional requirements. Rules Manager addresses these requirements by:

- Aggregating events for rule execution

When two or more primitive events are brought together, each primitive event may be generated by the application at different points in time. This often means a rule cannot be evaluated conclusively until all the primitive events are available. Rules Manager manages the primitive events and joins them together before evaluating the rules. Rules Manager hides the complexity of managing composite

events by maintaining the association between the primitive events and the composite event. See [Chapter 4](#) for more information.

- Maintaining intermediate state of event processing

When composite events are completely formed in the user application, some parts of rule conditions may need to be evaluated repeatedly with some parts of the composite events. This may lead to multiple evaluations of one primitive event for each instance of a second primitive event, and so forth to find matching rules. This evaluation becomes complex very quickly as the number of primitive events exceeds two. XML tags support incremental evaluation of rules for composite events resulting in Rules Manager improving the performance of the system. Rules Manager maintains the intermediate state of rule evaluation persistently for efficient processing. See [Section 4.1](#) for more information.

- Supporting complex rule constructs

Rules Manager enables you to build complex rules with negation, Any *n*, and Set semantics in conditional expressions. Using XML tags within rule conditions, Rules Manager can support these complex rule constructs that are commonly used in applications. See [Chapter 4](#) for more information.

2.4.1 How to Create a Rules Application That Uses Composite Events

The basic steps to create a rules application with composite events are the same as those described for simple events in [Section 2.3](#), with accommodations for multiple primitive events.

The steps to create a rules application with composite events are as follows:

1. Create the composite event structure as an object type in the database.

First, each primitive event structure is created as an object type. For example:

```
CREATE or REPLACE TYPE AddFlight AS OBJECT (
    CustId      NUMBER,
    Airline     VARCHAR2(20),
    FromCity    VARCHAR2(30),
    ToCity      VARCHAR2(30),
    Depart      DATE,
    Return      DATE);

CREATE or REPLACE TYPE AddRentalCar AS OBJECT (
    CustId      NUMBER,
    CarType     VARCHAR2(20),
    CheckOut    DATE,
    CheckIn     DATE,
    Options     VARCHAR2(30));
```

Next, all the primitive event structures that constitute the composite event are created as (first level) embedded types in this object type. For example:

```
CREATE or REPLACE TYPE TSCompEvent AS OBJECT (Flt AddFlight,
                                             Car AddRentalCar);
```

The attribute names, *Flt* and *Car*, are used in the rule conditions for identifying the predicates on individual primitive events and for specifying join conditions between primitive events; *Flt* and *Car* are the primitive event variables used for composite events.

2. Create the rule class for the composite event structure. The rule class is configured for composite events using an XML properties document that is assigned to the properties argument of the `dbms_rlmgr.create_rule_class` procedure.

```
BEGIN
  dbms_rlmgr.create_rule_class (
    rule_class    => 'CompTravelPromo',
    event_struct  => 'TSCompEvent',
    action_cbk    => 'CompPromoAction',
    rslt_viewnm   => 'CompMatchingPromos',
    actprf_spec   => 'PromoType VARCHAR2(20),
                   OfferedBy VARCHAR2(20)',
    rlcls_prop    => '<composite/>');
END;
```

The previous code example creates the rule class for the composite event structure. The `rlcls_prop` argument specifies the default settings for composite event management using an empty `<composite>` XML element. This step re-creates each object type representing a primitive event structure to include a timestamp attribute, `rlm$CrtTime`, which captures the corresponding event creation times. This attribute is created with the `TIMESTAMP` data type and its value is defaulted to the database timestamp (`SYSTIMESTAMP`) at the time of event instantiation. Alternately, an application can explicitly set an event creation time by assigning a valid timestamp value to this attribute.

As previously mentioned, this rule class creation also creates the action callback procedure with the specified name as follows:

```
PROCEDURE CompPromotion (Flt      AddFlight,
                        Car       AddRentalCar,
                        rlm$rule   CompTravelPromo%ROWTYPE) is
BEGIN
  null;
  --- The action for the matching rules can be performed here.
  --- The appropriate action can be determined from the event
  --- instance and the action preferences associated with each rule.
END;
```

Note: The primitive events within the composite events are passed in as separate arguments to the callback procedure.

3. Replace the system generated action callback procedure with the user implementation to perform the appropriate action for each matching rule. For example:

```
PROCEDURE CompPromoAction (Flt      AddFlight,
                          Car       AddRentalCar,
                          rlm$rule   CompTravelPromo%ROWTYPE) is
BEGIN
  OfferPromotion (Flt.CustId,
                 rlm$rule.PromoType,
                 rlm$rule.OfferedBy);
END;
```

4. Add the rules to the rule class. In this case, add a rule with a conditional expression that uses XML tags. See [Section 4.1](#) for more information about using XML tag extensions in rule conditions to support complex rule constructs.

```

INSERT INTO CompTravelPromo (rlm$ruleid, PromoType, OfferedBy, rlm$rulecond)
VALUES ('UN-HT-FL', 'RentalCar', 'Acar',
       '<condition>
         <and join="Flt.CustId = Car.CustId">
           <object name="Flt">
             Airline='Abcair' and ToCity='Orlando'
           </object>
           <object name="Car">
             CarType = 'Luxury'
           </object>
         </and>
       </condition>');
    
```

5. Process the rules using one primitive event at a time. For example:

```

BEGIN
  dbms_rlmgr.process_rules (
    rule_class => 'CompTravelPromo',
    event_inst =>
      AnyData.ConvertObject(
        AddFlight(987, 'Abcair', 'Boston', 'Orlando',
                  '01-APR-2003', '08-APR-2003'));

  dbms_rlmgr.process_rules (
    rule_class => 'CompTravelPromo',
    event_inst =>
      AnyData.ConvertObject(
        AddFlight(567, 'Abdair', 'Boston', 'Miami',
                  '03-APR-2003', '09-APR-2003'));

  dbms_rlmgr.process_rules (
    rule_class => 'CompTravelPromo',
    event_inst =>
      AnyData.ConvertObject(
        AddRentalCar(987, 'Luxury', '03-APR-2003',
                     '08-APR-2003', NULL));

END;
    
```

This command adds three primitive events to the Rules Manager. For the rule defined in Step 4, the first event matches the primitive rule condition for the `AddFlight` event and the third event matches the condition for the `AddRentalCar` event. Additionally, these two events satisfy the join predicate in the rule condition. So for the previous example, the first and last primitive events together form a composite event that matches the rule condition specified in Step 4. These primitive event instances are passed to the action callback procedure for action execution. The type information for the primitive events that is passed in is embedded in the corresponding `AnyData` instance. However, when a string-formatted event is used, the primitive event type information should be explicitly passed in as follows:

```

BEGIN
  dbms_rlmgr.process_rules (
    rule_class => 'TravelPromotion',
    event_type => 'AddFlight',
    event_inst =>
      AddFlight.getVarchar(987, 'Abcair', 'Boston', 'Orlando',
                           '01-APR-2003', '08-APR-2003'));

END;
    
```

2.4.2 Evaluating Composite Events Using Complex Rule Conditions

Evaluating composite events using complex rule conditions is supported by Rules Manager with the following:

- Incremental evaluation of rules by allowing predicate joins between and among primitive events
- Negation in rule conditions to raise exceptions in processes (that is, when something does not happen, do something)
- Sequencing in rule conditions by tracking primitive event creation time and enforcing or detecting sequencing among events
- Set semantics in rule conditions to allow instances of primitive events of the same type to be monitored as a group
- Any n in rule conditions to allow matching of a subset of primitive events

Rules Manager supports incremental evaluation of rules involving composite events. To support complex rule conditions, the conditional expressions in the SQL `WHERE` clause are extended with some XML tags that identify different parts of a conditional expression and add special semantics to these expressions. [Chapter 4](#) describes more about each type of complex rule condition. [Section 4.1](#) describes implementing incremental evaluation of rules.

2.5 Setting Event Management Policies (Rule Class Properties) for Rule Applications

Rule class properties define the event management policies that the Rules Manager should enforce for each rule application. Rule class properties include:

- Consumption -- determines if an event can be used for multiple rule executions or a single rule execution
- Conflict resolution or ordering -- determines the order in which matching rules with various events are to be executed
- Duration -- determines the lifetime of unconsumed primitive events
- Auto-commit -- determines if each interaction with a rule class should be committed automatically
- Storage -- determines the storage characteristics of the rule class in the database
- Equal -- specifies the common equality join predicates for all the rules in a rule class, that is, what are the lists of primitive event attributes that are equal in the composite events configured for a rule class
- DML Events -- specifies when an event structure is created with one or more table alias attributes, that the corresponding rule class should consider the data manipulation language (DML) operations (`INSERT`) on the corresponding tables as the events for which the rules are evaluated

Rule class properties are specified at the time of rule class creation using an XML properties document that is assigned to the `rlcls_prop` argument of the `dbms_rlmgr.create_rule_set()` procedure. For rule classes configured for composite events these properties can be specified at the composite event level (for all the primitive events). In addition, you can specify overrides for one or more primitive events in the properties document. [Section 3.1](#) through [Section 3.7](#) describe each of these rules properties in more detail and how each is implemented.

2.6 Creating Rules Applications That Span Multiple Tiers

For rules applications that span multiple tiers and where rule management is handled in the database, but the action execution for the rules is handled in the application server, the actions for the rules matching an event cannot be invoked from an action callback procedure. Instead, a results view is populated with the events and the matching rules, both of which are available for external action execution. The results view can be queried to determine the rules that match an event and their corresponding actions can then be executed.

To handle rules applications with certain rules having their action execution occurring on the application server, you must also configure the rule class for external execution (in addition to configuring the action callback procedure). The steps to do this are similar to those described in [Section 2.3](#), but are modified and briefly described as follows (see [Chapter 5](#) for a complete description of each step):

1. Create the event structure as an object type in the database (same as Step 1 in [Section 2.3](#)).
2. Create the rule class and also define the results view. See Step 2 in [Section 5.1](#) for the details.
3. Implement the action callback procedure (same as Step 3 in [Section 2.3](#)).
4. Add rules to the rule class (same as Step 4 in [Section 2.3](#)).
5. Identify the matching rules for an event. Use the add event procedure (`dbms_rlmgr.add_event()`) that adds each event to the rule class one at a time and identifies the matching rules for a given event that is later accessed using the results view. See Step 5 in [Section 5.1](#) for the details.
6. Find the matching rules by querying the results view. See Step 6 in [Section 5.1](#) for the details.
7. Consume the event that is used in a rule execution. See Step 7 in [Section 5.1](#) for the details.

For more information about creating rules applications that span multiple tiers, see [Section 5.1](#), and for more information about running rules applications in multitier mode see [Section 5.2](#).

2.7 Using Rules Manager with SQL*Loader and Export/Import Utilities

[Section 2.7.1](#) describes using SQL*Loader to load data into a rule class table.

[Section 2.7.2](#) describes exporting and importing rules applications.

2.7.1 SQL*Loader

SQL*Loader can be used to bulk load data from an ASCII file into a rule class table. For the loader operations, the rule conditions stored in the `rlm$rulecond` column of the rule class table are treated as strings loaded into a `VARCHAR2` column. The data file can hold the XML and SQL based rule conditions in any format allowed for `VARCHAR2` data and the values for the action preference columns in the rule class table are loaded using normal SQL*Loader semantics.

The data loaded into the rule condition column is automatically validated using the event structure associated with the rule class. The validation is done by a trigger defined on the rule condition column, due to which, a direct load cannot be used while loading rule definitions into a rule class table.

2.7.2 Export/Import

A rules application defined using a set of event structures and a rule class can be exported and imported back to the same database or a different Oracle database. A rule class in a schema is automatically exported when the corresponding rule class table is exported using the export command's (`expdp`) `tables` parameter or when the complete schema is exported. When a rule class is exported, definitions for the associated primitive and composite event structures and the rules defined in the rule class are all placed in the export dump file. However, the internal database objects that maintain the information about event instances and incremental states for partially matched rules are not exported with the rule class. When the `tables` parameter is used to export a particular rule class, the implementation for the action callback procedure is not written to the export dump file. The action callback procedure is only exported with the schema export.

The dump file created with the export of a rule class can be used to import the rule class and its event structures into the same or a different Oracle database. At the time of import, the internal database objects used for the rule class state maintenance are re-created. Due to the order in which certain objects are created and skipped in an import session, the rule class creation raises some errors and warnings that can be safely ignored. In the case of a schema level import of the rule class, the implementation for action callback procedure is also re-created on the import site. However, in the case of a table-level import, only the skeleton for the action callback procedure is created.

Event Management Policies

The rule class properties specified at the time of creation include the event management policies that the Rules Manager should enforce for each rules application. In the case of rules defined for composite event structures, the primitive events are added to the system one at a time. These events are later combined with other primitive events to form composite events that match one or more rule conditions. Depending on the join conditions between primitive events, a primitive event can participate in a 1 to 1, 1 to N , or N -to- M relationship with other events to form one or more composite events. Application-specific requirements for reusing primitive events and for handling duplicate composite events are supported using rule class properties and they are broadly classified as follows:

- Consumption -- determines if an event can be used for multiple rule executions or a single rule execution
- Conflict resolution or ordering -- determines the order in which matching rules with various events are to be executed
- Duration -- determines the lifetime of unconsumed primitive events
- Auto-commit -- determines if each interaction with a rule class should be committed automatically
- Storage -- determines the storage characteristics of the rule class in the database
- Equal -- specifies the common equality join predicates for all the rules in a rule class, that is, what are the lists of primitive event attributes that are equal in the composite events configured for a rule class
- DML Events -- specifies when an event structure is created with one or more table alias attributes, that the corresponding rule class can be configured to consider the DML operation (`INSERT`) on the corresponding tables as the events for which the rules are evaluated

The event management policies `duration` and `equal` are only applicable to rule classes configured for composite events. All other policies are applicable to rule classes configured for simple events as well as rule classes configured for composite events. Each rule class property is described in the sections that follow.

3.1 Consumption of Events

A primitive event used to form a composite event can be combined with other primitive events to form a different composite event. For example, two instances of the `AddFlight` event can be combined with one instance of `AddRentalCar` event to form two different composite events (that could match two different rules). In some rule applications, it is required that once a primitive event matches a rule on its own or

in combination with other primitive events, it should not be used with any more rule executions. This implies that a primitive event used to form a composite event is consumed or removed from the system. The `consumption` property for a rule class determines the policy regarding the reuse of primitive events in the system. The consumption policy is applicable to both the rules defined for simple events and the rules defined for composite events. Two modes of event consumption are possible:

- **EXCLUSIVE** -- when the consumption mode is `EXCLUSIVE`, a primitive event can be used to match only one rule (which ever matches first). Once the corresponding rule action is executed, this event is removed from the system, irrespective of the event duration specification (`TRANSACTION`, `SESSION`, or `Elapsed time`).
- **SHARED** -- when the consumption mode is `SHARED`, a primitive event can be used to match any number of rules and execute their actions. The primitive event is removed from the system only when its duration specification is met. The default consumption policy for a rule class created with no consumption property is `SHARED`.

Following the same example used previously, if two `AddFlight` events are already added to the system, the next `AddRentalCar` event could form two composite events that could match two or more rules. If the rule class is configured for `EXCLUSIVE` consumption of events, only one of the rule actions can be executed using one of the composite events. This rule can be chosen deterministically if appropriate conflict resolution techniques are employed (see [Section 3.2](#)).

The `EXCLUSIVE` consumption policy for a rule class created for a simple event structure implies that, at most, one rule is executed for any event instance passed to the `dbms_rlmgr.process_rules` procedure. If the event matches more than one rule, the rule that is chosen for execution is determined using the ordering property of the rule class (see [Section 3.3](#) that describes ordering). The rule class created for a primitive event structure can be configured with the `EXCLUSIVE` event consumption policy using the following XML properties document (as the `rlcls_prop` argument to the `dbms_rlmgr.create_rule_class` procedure).

```
<simple consumption="exclusive"/>
```

Other valid forms of consumption specification within the rule class properties include the following:

```
<composite consumption="exclusive"/>
```

```
<composite consumption="shared"/>
```

Rule applications can have different policies regarding the reuse of primitive events for a subset of primitive events used in a composite event. For such applications, the consumption policy can be specified for each primitive event type as a child element of the `<composite>` element, such as the following:

```
<composite consumption="shared">  
  <object type="AddFlight" consumption="shared">  
  <object type="AddRentalCar" consumption="exclusive">  
</composite>
```

The value for the `consumption` attribute of the `<composite>` element is used as the default value for all the primitive events in the composite event. This default value is overridden for a primitive event type by specifying it as the child element of the `<composite>` element and specifying the consumption attribute for this element.

Specifying Custom Logic for Event Consumption

In addition to `EXCLUSIVE` and `SHARED` consumption policies, a rule class for composite events can be configured with a `RULE` consumption policy, which allows individual rules in the rule class to use some custom logic for event consumption. The `RULE` consumption policy can only be specified at the composite event level and when specified, the consumption policy for the primitive event type cannot be set to `EXCLUSIVE`. When the rule class is configured for `RULE` consumption policy, the action callback procedure and the rule class results view are created to return the identifiers for the individual primitive events matching a rule. These identifiers can be used to selectively consume some or all of the primitive events atomically. See the `DBMS_RLMGR.CONSUME_PRIM_EVENTS` procedure for more information.

3.2 Conflict Resolution

When an event matches a rule on its own or in combination with other primitive events, by default, the order of rule (action) executions is not deterministic. Some rule applications may need the matching rules to execute in a particular order, which is determined by some conflict resolution criteria. Additionally, in the case of exclusive consumption of events, only one of the matching rules is executed. Unless some conflict resolution criterion is specified, this rule is chosen randomly. One of the common techniques of conflict resolution is to order the resulting composite events and matching rules based on the event attribute values and the rule action preferences.

- Conflict resolution among composite events

The composite events resulting from the addition of a primitive event can be ordered based on the attributes of the corresponding primitive events. For example, the travel services application may decide to resolve among a set of composite events, consisting of `AddFlight` and `AddRentalCar` primitive events, based on the primitive event creation times. So, the conflict resolution criterion for this composite event structure is represented as `[Flt.rlm$CrTTime, Car.rlm$CrTTime]`, implying that an event with earliest creation time is consumed before the others. This notation is similar to that of an `ORDER BY` clause in a SQL query. Optionally, the `DESC` keyword can be used with some of the attributes to sort the events in descending order (see [Section 3.3](#) for complete syntax). When the rule class is configured for exclusive consumption of events, only the top-most event in this sorted list is chosen for rule execution.

- Conflict resolution among matching rules for simple and composite events

A composite or a simple event can match one or more rules in a rule class. If more than one rule is matched, by default, their actions are executed in a non-deterministic order. If the order of the rule action executions is important, the rule identifiers and the action preferences associated with the rules can be used to sort the matching rules. For example, the travel services application can resolve among matching rules by using the conflict resolution criterion - `[rlm$rule.PromoType, rlm$rule.OfferedBy, rlm$rule.rlm$ruleid]`. In this case, sorting the matching rules is done in ascending order and ordered first by the action preference `PromoType`, then by the action preference `OfferedBy`, then by the rule identifier `rlm$ruleid`. As shown in this example, `rlm$rule` is used to refer to any rule-specific attribute. The notation and semantics used for specifying conflict resolution criteria are similar to that of an `ORDER BY` clause in a SQL query. Optionally, the `DESC` keyword can be used with some of the attributes to sort the rules in the descending order (see [Section 3.3](#) for a complete syntax). When the rule class is configured for exclusive consumption of events, only the top-most rule in the sorted list is chosen for execution.

When a set of composite events matches a set of rules, the exact order of rule executions can be specified by combining the conflict resolution criterion for the composite events with that of the matching rules. The syntax for specifying the conflict resolution criteria is described in [Section 3.3](#).

3.3 Ordering of Rule Execution

The `ORDERING` property of the rule class determines the order in which a set of rules that match a set of composite events or a simple event are executed. When the consumption policy for a composite event type or for some primitive event types is set to `EXCLUSIVE`, the ordering property also determines the subset of rules that are executed. (The rest of the matching rules are ignored, because the exclusive events that are required to execute the rules are deleted after the first rule execution). The ordering property is applicable to both the rules defined for simple events and the rules defined for composite events.

In the case of a rule class created for a composite event structure, the addition of a primitive event to the system could form multiple composite events that could match multiple rules. So, the ordering of the resulting events and the matching rules can be specified using the attributes in the events, the action preferences associated with the rules, and the rule identifiers. For the travel services rule class example, the ordering of the events and the matching rules can be specified as follows:

```
<composite ordering="Flt.rlm$CrtTime, Car.rlm$CrtTime, rlm$rule.PromoType,
rlm$rule.OfferedBy, rlm$rule.rlm$ruleid"/>
```

In this ascending column, attribute ranked, ordering specification, `rlm$rule` is used to refer to the attributes associated with the rule class (action preferences `PromoType` and `OfferedBy` and the rule identifier `rlm$ruleid`) and the variables declared for the primitive events in the composite event structure (`Flt` for `AddFlight` and `Car` for `AddRentalCar`) are used to access the primitive events' attribute values.

The ordering property can be combined with some other policies, such as consumption and duration. Other valid forms of ordering specification within the rule class properties include:

```
<composite consumption="exclusive"
      ordering="Flt.rlm$CrtTime, rlm$rule.PromoType,
              rlm$rule.rlm$ruleid DESC"/>
<simple ordering="rlm$rule.PromoType, rlm$rule.OfferedBy, rlm$rule.rlm$ruleid/>
```

In the case of a rule class created for a simple event structure, as there is only one event at any point in time, the ordering is only based on the matched rules. So, only the rule identifier and action preferences associated with the rules are allowed in the ordering clause.

3.4 Duration of Events

It is common for applications to generate events that will never trigger a rule firing, thus these events will never be consumed. The duration policy for primitive events determines the maximum lifetime of the events. When a primitive event is added to the Rules Manager for incremental evaluation of rules, the event and the evaluation results are stored in the database. These events are later combined with other matching primitive events to form a composite event that conclusively satisfies one or more rule conditions. However, there may not be a matching event to form a composite event. For example, the travel services rule discussed in [Section 2.6](#) may detect an `AddFlight` event for a rule, but the corresponding `AddRentalCar` event may not

occur (or the `AddRentalCar` event occurring may not be for a luxury car). So, the duration (or the life) of the primitive events should be set such that the incomplete (composite) events and the results from the incremental rule evaluations are deleted after a certain period.

The duration of a primitive event depends on the rule application and it can be classified into one of following four scenarios.

- **TRANSACTION:** In this scenario, the primitive events added to the system during a database transaction are preserved until the end of the transaction (`COMMIT` or `ROLLBACK`). So, a rule for the composite event evaluates to true only if all the required primitive events are detected within a database transaction.
- **SESSION:** In this scenario, the primitive events added during a database session are preserved until the end of the session (`CONNECT` or `DISCONNECT`). So, a rule for the composite event evaluates to true only if all the required primitive events are detected within a database session.
- **CALL:** In some rule applications, a subset of primitive events are truly transient in that an event is considered for a possible match with the rules only at the instance at which the event is added. Such events do not contribute to the event history and they are not considered for any future rule matches. Hence, these events are said to be valid only for the duration of the call (`PROCESS_RULES` or `ADD_EVENT`) that processes the rules. A subset of primitive events within a composite event can be configured for the `CALL` duration. A `CALL` duration event contributes to a rule execution only if the event, in combination with other events in the system, evaluates a rule condition to true at the time of the call. Such events are not considered for rule matches after the call regardless of any rule executions during the call.
- **Elapsed time:** In this scenario, the duration of a primitive event added to the system is determined by an event timeout associated with the rule class. The event timeout is specified as elapsed time (for example 10 hours, 3 days) and this is added to the creation time (determined by its `rlm$CrTTime` attribute) to determine the exact time of event deletion.

The duration policy dictates the life span of the primitive events in the system. In the case of a rule class created for simple events, Rules Manager does not store the events in the system (as the rules are evaluated conclusively for each event). So, the duration policy is applicable only for the rule classes created for composite event structures. A rule class configured to reset all the primitive events at the end of each (database) transaction uses the following XML properties document:

```
<composite duration="transaction"/>
```

While specifying the duration as elapsed time, the value for the duration attribute can be specified in `{[int] minutes | hours | days}` format, such as shown here:

```
<composite duration="session"/>
<composite duration="20 minutes"/>
<composite duration="2 hours"/>
<composite duration="10 days"/>
```

These specifications apply to all the primitive events that constitute the composite event. If different duration specifications are required for some primitive event types, they can be specified as child elements of the `<composite>` element, such as shown here:

```
<composite duration="10 days">
  <object type="AddFlight" duration="3 days"/>
  <object type="AddRentalCar" duration="call"/>
```

```
</composite>
```

In this case, the value of 10 days for the `duration` attribute of the `<composite>` element is used as the default value for all the primitive events in the composite event. This default value is overridden for a primitive event type by specifying it as the child element of the `<composite>` element and specifying the `duration` attribute for this element, for example, as shown by the `duration` property call specified for the `AddRentalCar` event type. So these `AddRentalCar` events would be discarded if they did not match a rule during either a `PROCESS_RULES` or `ADD_EVENT` call.

A restriction on the duration policy is that the `TRANSACTION` or `SESSION` values for the duration policy can only be specified at the composite event level. When specified, these values cannot be overridden at the primitive events level.

3.5 Equality

In rule applications using composite events, all the rules defined in a rule class have one or more common (equality) join predicates between two or more or all primitive events (that form a composite event). These join predicates are defined using the attributes of the corresponding primitive event types. For example, in the travel services application, the `AddFlight` and `AddRentalCar` events in a composite event are related through the customer identifiers in these primitive events (`Flt.CustId = Car.CustId`). The rule class using a fixed number of distinct equality join predicates for its rules can be optimized by configuring the rule class as discussed in this section.

Note: Use of the `EQUAL` property at the rule class level is recommended for better performance.

For a rule class configured for the composite event structure, the `EQUAL` property identifies either the only equality join predicate that relates multiple primitive events within each composite event, or a list of the most common equality join predicates used in forming each composite event. If there is only one equality join predicate, the `equal` property is specified as a comma-delimited list of attributes, one from each primitive event structure configured for the rule class and it is used as the join predicate for all the rules in the rule class. This list identifies the primitive event attributes that must be equal to form a composite event. For example:

```
<composite equal="Flt.CustId, Car.CustId"/>
```

When the `EQUAL` property is specified as a comma separated list of attributes, the rules in the corresponding rule class may skip the equality join predicates involving the same set of attributes. So, for a rule class created with the previous properties document, the following three rule conditions have the same effect:

```
<condition>
  <and join="Flt.CustId = Car.CustId and Car.rlm$CrtTime > Flt.rlm$CrtTime ">
    <object name="Flt"> Airline='Abcair' and ToCity='Orlando' </object>
    <object> CarType = 'Luxury' </object>
  </and>
</condition>
```

```
<condition>
  <and join="Car.rlm$CrtTime > Flt.rlm$CrtTime">
    <object name="Flt"> Airline='Abcair' and ToCity='Orlando' </object>
    <object name="Car"> CarType = 'Luxury' </object>
  </and>
</condition>
```

```

<condition>
  <and equal="Flt.CustId, Car.CustId"
    join="Car.rlm$CrtTime > Flt.rlm$CrtTime">
    <object name="Flt"> Airline='Abcair' and ToCity='Orlando' </object>
    <object> CarType = 'Luxury' </object>
  </and>
</condition>

```

The last rule condition in the previous example also demonstrates the use of the EQUAL property for individual rules in a rule class. The EQUAL property specification for a rule condition acts as a short representation of the equality join predicate, especially when the rule condition has ANY *n* or negation constructs. The list of attributes specified for the EQUAL property are converted into appropriate join predicates and appended (as conjunctions) to the join condition of each rule in the rule class.

Another form of EQUAL property specification for a rule class identifies a list of the most common equality join predicates in its rules. For this purpose, each EQUAL property specification is grouped using parenthesis and alternate EQUAL property specifications are separated using a vertical bar ('|') character. For example, in a rule class created with two primitive events of the same RFIDRead type, a subset of rules in the rule class may join the primitive events on their ItemId attributes (reading1.ItemId = reading2.ItemId). Another subset of the rules in the same rule class may relate the primitive events on their ReaderId attributes (reading1.ReaderId = reading2.ReaderId). The rule class can be optimized to process both types of these rules efficiently using the following equal property:

```

<composite equal="(reading1.ItemId, reading2.ItemId) |
  (reading1.ReaderId, reading2.ReaderId)"/>

```

The previous syntax for the EQUAL property can be used to specify up to five alternate EQUAL property specifications for a rule class. The alternate EQUAL property specifications provide a means for optimizing the rule evaluation for the most common join predicates in a rule class and it does not automatically use any equality join predicates for its rules. The rules in the rule class are mapped into one of the alternate EQUAL property specifications based on the EQUAL property used for the individual rules. For example, the following rule's EQUAL property specification matches one of the EQUAL property specifications at the rule class level and hence, this rule is optimized:

```

<condition>
  <and equal="reading1.ItemId, reading2.ItemId"/>
  <object name="reading1"/>
  <object name="reading2"/>
</and>
</condition>

```

Hence, the EQUAL property for individual rules in a rule class not only acts as a short representation for the equality join predicate but also helps map it into one of the alternate EQUAL property specifications.

3.6 Storage Properties

The STORAGE attribute of the <simple> or <composite> element is used to specify the storage properties for the rule class table and the internal objects created for the rule class. By default, the database objects used to manage the rules in a rule class are created using user defaults for the storage properties (Example: tablespace

specification). The value assigned for this attribute can be any valid storage properties that can be specified in a typical SQL `CREATE TABLE` statement. The following XML properties document can be used (as the argument to the `dbms_rlmgr.create_rule_class` procedure) to create a rule class for simple events that resides in a tablespace `TBS_1` and uses exclusive consumption policy:

```
<simple storage="tablespace TBS_1" consumption="exclusive"/>
```

An example of specifying storage attributes in the rule class properties is as shown:

```
<composite storage="tablespace TBS_1"/>
```

3.7 AUTOCOMMIT

In most cases, all the Rules Manager procedures commit immediately after each add rule, delete rule, and process rule operation. The rule class can be configured to follow transaction boundaries by turning off the auto-commit functionality. For this purpose, the `AUTOCOMMIT` property can be specified in the rules class properties document. For example:

```
<simple autocommit="NO"/>
```

The `AUTOCOMMIT` property can be specified for the rule class created for simple as well as composite events. Other valid forms of specifying the `AUTOCOMMIT` property include:

```
<composite autocommit="NO" consumption="shared"/>  
<composite autocommit="YES"/>
```

When the `AUTOCOMMIT` property is set to `NO`, the set of Rules Manager operations (add rule, delete rule, and process rule) performed in a transaction can be rolled back by issuing a `ROLLBACK` statement. An exception to this rule is when the action callback procedure (implemented by the end user) performs an irreversible operation (sending a mail, performing a data definition language (DDL) operation, commit, rollback, and so forth). A DDL operation within an action callback operation automatically commits all the operations performed in that transaction. To avoid this situation, any DDL operations should be performed in an autonomous transaction.

Turning off the `AUTOCOMMIT` property for a rule class can limit the concurrent operations on the rule class. This is especially the case when the rules class is created for composite events that is configured for exclusive consumption policy. (In a transaction, the consumed events are locked until the transaction is committed and the other sessions may wait for these events to be released.)

The default value for the `AUTOCOMMIT` property is dependent on other event management policies (see [Table 3-1](#)). The default value for this policy is `NO` for a rule class configured for simple (non-composite) rules and a composite rule class configured with the `SESSION` or `TRANSACTION` duration policy. (These configurations do not pose issues with sharing of events across sessions). For all other configurations, a default value of `YES` is used for the `AUTOCOMMIT` property. Note that the `AUTOCOMMIT` property cannot be set to `YES` when the duration policy is set to `TRANSACTION`. Also, the `AUTOCOMMIT` property cannot be set to `NO` when one or more primitive event types are configured for `EXCLUSIVE` or `RULE` consumption policy.

When the event structure is defined with one or more table alias constructs and the corresponding rule class is configured for DML events (see [Section 3.8](#)) the `AUTOCOMMIT` property is always set to `NO`. Note that this could result in deadlocks while working with `EXCLUSIVE` or `RULE` consumption policies.

A rule class with the `AUTOCOMMIT` property set to "NO" cannot contain rules involving negation and a deadline (See [Section 4.3](#)).

3.8 DML Events

When an event structure is created with one or more table alias attributes (see [Section 3.10](#)), then the corresponding rule class can be configured to consider the SQL `INSERT` and `SQL*Loader` operations on the corresponding tables as the events for which the rules are evaluated. This rule class behavior can be enabled using the `DMLEVENTS` property for the rule class:

```
<simple dmlevents="I"/>
```

This property can be specified for a rule class configured for simple and composite events. In the current release, `UPDATE` and `DELETE` operations on the underlying tables are ignored and only the `INSERT` operations are recognized as events.

```
<composite dmlevents="I"/>
```

When the `DMLEVENTS` policy is specified, the `AUTOCOMMIT` policy for the rule class should be `NO`. In this case, the `AUTOCOMMIT` policy of `NO` is allowed even when the consumption policy is set to `EXCLUSIVE` or `RULE` (which is considered an invalid combination when the `DMLEVENTS` policy is not used). Note that the use of the `EXCLUSIVE` or `RULE` consumption policy with the `DMLEVENTS` policy could result in application deadlocks.

3.9 Rule Class Property Dependencies and Defaults

Most of the rule class properties (or event management policies) described in this section can be mixed and matched while defining a rule class. However, some of the combinations of these properties are considered invalid as shown in [Table 3-1](#). For example, if the rule classes' `AUTOCOMMIT` property is set to `YES`, setting the `DURATION` policy to `TRANSACTION` is invalid. This is because the events are deleted from the system as soon as they are added and they cannot be combined with other events to form composite events. The `DMLEVENTS` policy has no direct influence on the valid and invalid combination of event management policies. This policy only effects the default value for the `AUTOCOMMIT` policy.

Table 3-1 Valid and Invalid Rule Class Property Combinations

	AUTOCOMMIT	CONSUMPTION	DURATION
Invalid	Yes	--	Transaction
Valid	Yes	--	Session
Valid	Yes	--	[n] Units
Valid	No	Shared	--
Valid	No	Exclusive	Transaction ¹
Valid	No	Exclusive	Session ¹
Invalid	No	Exclusive	[n] Units ²
Valid	No	Rule ³	Transaction ¹
Valid	No	Rule	Session ¹
Invalid	No	Rule	[n] Units ²

- ¹ A rule class operating in `SESSION` or `TRANSACTION` mode has no concurrency issues across the database session, as each session gets a private copy of the events and incremental results.
- ² A rule class with the `EXCLUSIVE` consumption policy locks some rows in order to mark them "consumed" and may not actually consume the rows. Such rows are kept from being consumed by other database sessions and thus result in deadlocks. So, it is recommended that the locked rows be released with `AUTOCOMMIT="YES"` property.
- ³ `RULE` is a special form of the `EXCLUSIVE` consumption policy where the consumption of certain events is initiated by the end-user.

The default values for various event management policies for a rule class configured for simple events are as follows:

```
CONSUMPTION    : Shared
DURATION       : Infinite Duration (NULL)
AUTOCOMMIT     : No
```

The default values for the event management policies for a rule class configured for a composite event is sometimes dependent on other event management policies, as follows:

```
CONSUMPTION    : Shared
DURATION       : Infinite Duration (NULL)
AUTOCOMMIT
  IF DMLEVENTS = I                               : NO
  ELSE IF DURATION = TRANSACTION / SESSION       : NO
  ELSE                                             : YES
```

3.10 Rules Specified on Relational Tables

The rule applications considered so far use rule conditions that are defined on some application data. That is, the concept of an event instance exists in the application and it may or may not be stored in the database. Often however, the data in the event instances correspond to some rows stored in relational tables. For such applications, the row identifiers (ROWIDs) of these rows can be used to pass the data to the Rules Manager procedures by reference, for example using the `event_inst` parameter to represent an event instance of the `PROCESS_RULES` call (`event_inst => :FlightDataRowid`). For this purpose, the corresponding event structure should be modeled using Expression Filter's table alias constructs. See the `ADD_ELEMENTARY_ATTRIBUTE` procedure for more information. See [Section 9.2](#) and [Appendix A](#) for more examples.

For the travel services application considered in [Section 2.4](#), if the `AddFlight` and `AddRentalCar` primitive events are stored in two relational tables `FlightData` and `RentalCarData` respectively, the corresponding composite event structure can be created to refer to rows in these tables as follows:

```
BEGIN
  dbms_rlmgr.create_event_structure (event_structure => 'TSCompEvent');
  dbms_rlmgr.add_elementary_attribute (
    event_structure => 'TSCompEvent',
    attr_name => 'Flt', --- Prim event name
    attr_type => exf$table_alias('FlightData'));
  dbms_rlmgr.add_elementary_attribute (
    event_structure => 'TSCompEvent',
    attr_name => 'Car', --- Prim event name
    attr_type => exf$table_alias('RentalCarData'));
END;
```

Now the composite event structure `TSCompEvent` can be used to configure a rule class (Same as step 2 in [Section 2.4.1](#)). The representation of the rules in the rule class does not vary with this event structure. However, within the action callback procedure, the primitive event instances that match a rule are each passed in as ROWIDs from the corresponding tables and these ROWIDs can be used to obtain the original event data. Also, with this new event structure, the primitive event instances, for which the rules are processed, are passed in by reference using the ROWIDs for the corresponding table rows.

```
BEGIN
  dbms_rlmgr.process_rules (
    rule_set_nm => 'TravelPromotion',
    event_type => 'FlightData',
    event_inst => :FlightDataRowid); -- rowid of a row ---
END;
```

Note that the conditions corresponding to a primitive event are evaluated when the `dbms_rlmgr.process_rules` procedure is invoked with the appropriate ROWID. However, Rules Manager does not keep track of any changes to the original row through an UPDATE operation.

Optionally, the `PROCESS_RULES` call can be eliminated by configuring the preceding rule class to consider all the INSERT operations into the corresponding tables as events. This is done by setting the `DMLEVENTS` property to "I" during rule class creation (see [Section 3.8](#)). This specification automatically creates a few statement triggers that make the necessary `PROCESS_RULES` calls for the rule class.

When the duration and consumption policies are set for the primitive events derived from relational data, it is the references to the table rows that are consumed or deleted from the rule class. The original rows are not affected by these event management policies.

A composite event structure can be formed using a combination of table alias constructs and embedded abstract data types (ADTs) (for various primitive events). The rule conditions defined for a composite event structure consisting of one or more table alias constructs may not use the (short form of) the `SEQUENCE` property to enforce sequencing among primitive events (see [Section 4.2](#)). This is because the implicit attribute `rlm$crtime` may not exist for the rows stored in the relational tables. The user can enforce partial sequencing using the `join` property in the rule conditions.

3.11 Rules Conditions For XML Events

The `XMLType` data type supplied by Oracle can be used to create attributes in the event structures and rule classes that can process rules defined on XML documents. For this purpose, a primitive event structure can be created with one or more `XMLType` attributes (along with some non-XML attributes), such as the following:

```
CREATE or REPLACE TYPE AddFlight AS OBJECT (
  CustId NUMBER,
  Airline VARCHAR(20),
  FromCity VARCHAR(30),
  ToCity VARCHAR(30),
  Depart DATE,
  Return DATE,
  Details sys.XMLType)
```

If a primitive event is just an XML document, then the preceding object type can be created with just one attribute, that is of `XMLType`. The predicates on the `XMLType`

attributes are specified using the `EXTRACT` and `EXISTS` operators supplied by Oracle, as shown in the following example.

```
<condition> <!-- optional for conditions on primitive events -->
  Airline='Abcair' and ToCity='Orlando' and
    EXTRACT(doc, '/preferences/seat[@class="economy"]') is not null
</condition>
```

A composite event structure for XML events can be formed by including two or more primitive event types that contain an `XMLType` attribute. So, the composite event structure is created as an object type with embedded primitive event types (as described in [Section 2.4](#)). Once the event structures are created with `XMLType` attributes, all the other concepts described in [Section 2.2](#) apply to the XML data that is part of the events.

3.12 Rule Conditions with Spatial Predicates

Note: The Oracle Spatial or the Locator components must be installed in order to use spatial predicates in stored expressions.

The `SDO_GEOMETRY` data type supplied by Oracle can be used to create event structures and rule classes that can process rules defined on spatial geometries. For this purpose, a primitive event structure can be created with one or more attributes of the `MDSYS.SDO_GEOMETRY` type, as follows:

```
CREATE or REPLACE TYPE AddHotel AS OBJECT (
  CustId    NUMBER,
  Type      VARCHAR(20),
  CheckIn   DATE,
  CheckOut  DATE,
  Location  MDSYS.SDO_GEOMETRY)
```

In order to specify predicates on the spatial attributes and index them for efficiency, the geometry metadata describing the dimension, lower and upper bounds, and tolerance in each dimension should be associated with each spatial geometry attribute. This metadata information can be inserted into the `USER_SDO_GEOM_METADATA` view using the event structure name in the place of the table name. For more information on the `USER_SDO_GEOM_METADATA` view and its semantics, see *Oracle Spatial User's Guide and Reference*.

```
INSERT INTO user_sdo_geom_metadata VALUES ('ADDHOTEL', 'LOCATION',
  mdsys.sdo_dim_array(
    mdsys.sdo_dim_element('X', -180, 180, 0.5),
    mdsys.sdo_dim_element('Y', -90, 90, 0.5)), 8307);
```

When the event structure with spatial attributes is used to create a rule class, the rule conditions stored in the rule class table can include predicates in these attributes using `SDO_WITHIN_DISTANCE` or `SDO_RELATE` operators, as shown in the following examples:

```
<condition>
  Type = 'Luxury' and CheckOut-CheckIn > 3 and
    SDO_WITHIN_DISTANCE (Location,
      SDO_GEOMETRY(2001, 8307,
        SDO_POINT_TYPE(-77.03644, 37.89868, NULL), NULL, NULL),
      'distance=0.5 units=mile') = 'TRUE'
</condition>
```

```

<condition>
  Type = 'Luxury' and CheckOut-CheckIn > 3 and
  SDO_RELATE (Location,
    SDO_GEOMETRY(2001, 8307, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 3),
      SDO_ORDINATE_ARRAY(-77.03644, 37.89868, -75, 39),
    'mask=anyinteract') = 'TRUE'
</condition>

```

A composite event structure involving spatial attributes can be formed by including two or more primitive event structures that contain SDO_GEOMETRY attributes. In the case of rules specified for composite events, the spatial predicates involving SDO_WITHIN_DISTANCE or SDO_RELATE operators are not allowed in the join clause of the rule condition. If needed, functions defined in the MDSYS.SDO_GEOM package may be used to achieve this functionality. See *Oracle Spatial User's Guide and Reference* for additional information.

3.13 Database State in Rule Conditions

The predicates in a rule condition may use user-defined functions, references to database table data, and database state information. In the case of a condition specified for a simple or a primitive event, the corresponding predicates are evaluated using the state information at the time the event is added to the rule class (using either the PROCESS_RULES or the ADD_EVENT call. This is the case even for conditions specified for individual primitive events within a composite event. So, when multiple primitive events are used to capture a composite event, the predicates associated with individual primitive events will be evaluated at different times, corresponding to the occurrences of their respective event. Only the predicates specified in the join attribute of the composite condition are evaluated at the time of composite event creation. So, this aspect should be considered when using database state or schema object references in the rule conditions.

Rule Conditions

A rule condition is expressed using the attributes defined in the event structure. For the travel services example, the rule condition is expressed using the attributes: *Airline*, *ToCity*, *Return*, and *Depart*, as follows:

```
Airline = 'Abcair' and ToCity = 'Orlando' and Return - Depart >=7
```

Because a rule condition for a primitive event structure can be represented as a standard SQL *WHERE* clause conditional expression, the previous rule condition is shown without the XML markup. Were this rule condition to be represented with the XML markup, the expression would be as follows:

```
<condition>  
  Airline = 'Abcair' and ToCity = 'Orlando' and Return - Depart >=7  
</condition>
```

XML markup can be ignored in the case of rules defined for simple event structures because these cases usually involve simple usage. Because the rule condition specified for composite event structures involves complex constructs and must support incremental evaluation of rules, the condition expression is enhanced with some XML markup. This support is added as XML extensions to the conditional expressions that are typically in the SQL *WHERE* clause. The XML tags provided identify different parts of a conditional expression and add special semantics to this expression.

4.1 Support for Incremental Evaluation of Rules

To enable incremental evaluation of rules and support complex rule applications, the conditional expressions in the SQL *WHERE* clause are extended with meaningful XML tags. The XML tags are provided to identify different parts of a conditional expression and add special semantics to these expressions.

For example, the conditional expression (*Flt.Airline* = 'Abcair' and *Flt.ToCity* = 'Orlando' and *Flt.CustId* = *Car.CustId* and *Car.CarType* = 'Luxury') in the travel services rule has three parts, as follows:

- Predicates defined on the primitive event *AddFlight* (*Flt.Airline* = 'Abcair' and *Flt.ToCity* = 'Orlando')
- A predicate defined on the primitive event *AddRentalCar* (*Car.CarType* = 'Luxury')
- A join predicate between the two primitive events (*Flt.CustId* = *Car.CustId*)

Rules Manager provides XML tags to identify various parts of a complex conditional expression and support additional semantics. For example, the previous rule condition can be represented using XML tags as follows¹:

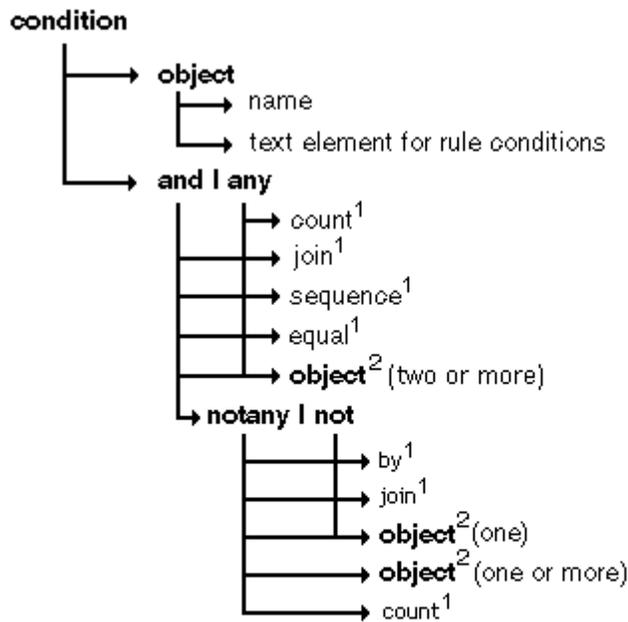
```
<condition>
  <and join="Flt.CustId = Car.CustId">
    <object name="Flt"> Airline='Abcair' and ToCity='Orlando' </object>
    <object name="Car"> CarType = 'Luxury' </object>
  </and>
</condition>
```

In this representation, the object elements capture the predicates specified for individual primitive events and one join attribute of the `<and>` element captures the join predicate behavior between two primitive events. The rule condition in this format can be inserted into the `rlm$rulecond` column of the corresponding rule class table. XML tags are provided to support more complex rule constructs. These tags are summarized in [Figure 4-1](#) and [Table 4-1](#) and are described in [Section 4.2](#) through [Section 4.5](#).

[Figure 4-1](#) describes a hierarchical view of the supported XML tag elements and their attributes for the rule condition XML schema definition that is described in detail in [Appendix F](#) in the Rule Condition section. [Table 4-1](#) shows a relational view of the same supported XML tag extensions showing the XPath and some notes about the elements and attributes.

¹ For simplicity, the examples in this document are shown without the XML entity references for `<` (`<`), `>` (`>`), and `'` (`'`) symbols. Within a rule condition, less than is the only symbol that must be specified using an entity reference (`<`) or used within a XML CDATA section. Although it is not mandatory to use entity references for other symbols, they are recommended for compatibility reasons. Following the SQL naming convention, all the values specified for XML attributes and elements are case-insensitive. Case is preserved only if a value appears within quotes.

Figure 4–1 Hierarchical View of the XML Tag Extensions



Bolded text represents elements; unbolded text represents attributes.

¹ An attribute can occur in any order and more than one can be used in any single rule condition.

² Represents the repeated attribute name and text element for rule conditions, which occurs where ever the **object** element appears.

Table 4–1 Relational View of the XML Tag Extensions

XML Tag	Type	Parent	XPath	Number of Occurrences Allowed within Its Parent	Notes
condition	Element	None	condition	---	Denotes a conditional expression
and	Element	condition	condition/and	One	Combines predicates
any	Element	condition	condition/any	One	A substitute for "or"; true if any condition is met
not	Element	and	and/not	One, as last child element	Logical negation
notany	Element	and	and/notany	One, as last child element	Logical negation; detects non-occurrence
object	Element	condition	condition/object	One	Primitive event
		and	and/object	Two or more objects	
		any	any/object	Two or more objects	
		not	not/object	One object	
		notany	notany/object	Two or more objects	

Table 4–1 (Cont.) Relational View of the XML Tag Extensions

XML Tag	Type	Parent	XPath	Number of Occurrences Allowed within Its Parent	Notes
join	Attribute	and	and/@join	One	Joins predicates
		any	any/@join	One	
		not	not/@join	One	
		notany	notany/@join	One	
sequence	Attribute	and	and/@sequence	One	Specifies an ordered sequence
		any	any/@sequence	One	Specifies any ordered sequence
equal	Attribute	and	and/@equal	One	Joins predicates
		any	any/@equal	One	
count	Attribute	any	any/@count	One	Any <i>n</i> semantics
		notany	notany/@count	One	Any <i>n</i> semantics
by	Attribute	not	not/@by	One	Deadline for non-occurrence
		notany	notany/@by	One	Deadline for non-occurrence
name	Attribute	object	object/@name	One	Object name

4.2 Rule Conditions with Sequencing

The rules defined for a composite event (consisting of two or more primitive events) may specify a condition on the order in which the primitive events should occur. This is called sequencing and it can be partial on a subset of primitive events, or it can be complete based on all the primitive events. Sequencing in rule applications is supported using the implicit timestamp attribute (`rlm$CrtTime`) that is included in each primitive event participating in a composite event.

The event creation times in the primitive events are used to enforce and detect sequencing in rule applications. For example, the rule considered in the travel services application can specify an additional predicate to offer the promotion only if the `AddRentalCar` event is generated after the `AddFlight` event. The rule condition can be extended to include this sequencing predicate, as follows:

```
<condition>
  <and join="Flt.CustId = Car.CustId" sequence="yes">
    <object name="Flt"> Airline='Abcair' and ToCity='Orlando' </object>
    <object name="Car"> CarType = 'Luxury' </object>
  </and>
</condition>
```

The sequence attribute in the preceding example ensures that the rule condition evaluates to true only if the matching primitive events occur in the order in which they are specified within the `<and>` element. The sequence attribute can be replaced with a join predicate on the corresponding event creation times, shown as follows:

```
<condition>
  <and join="Flt.CustId = Car.CustId and Car.rlm$CrtTime >= Flt.rlm$CrtTime">
    <object name="Flt"> Airline='Abcair' and ToCity='Orlando' </object>
    <object name="Car"> CarType = 'Luxury' </object>
  </and>
</condition>
```

```

    </and>
</condition>

```

Sequencing can be used to detect partial ordering among primitive events (for example, using a join predicate on only two primitive events when there are three of them in the composite event). The `rlm$CrtTime` attribute in the primitive event type can also be used to apply additional time constraints in the rule conditions. For example, the travel services rule may be valid only when the car reservations is made within 24 hours of making the flight reservation. This is indicated in the bolded text of the following example where the value 1 means one day. See *Oracle Database Application Developer's Guide - Fundamentals* for more information about performing date/timestamp arithmetic.

```

<condition>
  <and join="Flt.CustId = Car.CustId and
                                Flt.rlm$CrtTime >= (Car.rlm$CrtTime - 1)"
                                sequence="Yes">
    <object name="Flt"> Airline='Abcair' and ToCity='Orlando' </object>
    <object name="Car"> CarType = 'Luxury' </object>
  </and>
</condition>

```

Optionally, the call to the `DBMS_RLMGR.PROCESS_RULES` procedure may pass an event with a specific event creation time. Within a primitive event, the `rlm$CrtTime` attribute is treated as any other attribute in the event structure. However, when a value is not specified for this attribute, it is assigned a default value of `SYSTIMESTAMP` (in the database). If an application is sensitive to the difference between the times at which the events are detected (in the application layer) and the times at which they are added to Rules Manager, it may choose to set the values for event creation times and add fully specified events to the rules class.

4.3 Rule Conditions with Negation

Rules with negation in their conditions are typically used to raise exceptions in business processes. For example, a rule using negation could be "If an order is placed by a Gold customer and the items are not shipped within 24 hours of the order placement, alert the representative". In this case, the rule is defined for a composite event consisting of two primitive events `PlaceOrder` and `ShipOrder` and the type created for the composite event structure is shown as follows:

```

CREATE or REPLACE TYPE OrderTrack AS OBJECT (
    order PlaceOrder, -- primitive event type --
    ship ShipOrder); -- primitive event type --

```

For a composite event, a rule defined with negation evaluates to true when one of the primitive events does not happen within a time delay of the other. So, negation always accompanies a time delay that is relative to the other primitive event or events in the composite event. For example, the rule condition for the order tracking rule can be captured as follows, where the bolded text in the following example, "`sysdate +1`", means by the end of the next day because the SQL datetime function `SYSDATE` returns the current date and time of the operating system on which the database resides (taking into account the time zone of the database server's operating system that was in effect when the database was started).

```

<condition>
  <and equal="order.orderId, ship.orderId">
    <object name="order"> Type = 'Gold' </object>
    <not by="sysdate+1">

```

```

        <object name="ship"/> -- empty elem: no conditions on the primitive event --
    </not>
</and>
</condition>

```

The `<not>` XML element in the rule condition has the following semantics:

- There can be only one `<not>` element in a rule condition.
- The `<not>` element can only appear within an `<and>` element (as a conjunction to other primitive events) and it should be the last element within the `<and>` element.
- The `<not>` element is activated only when all the other primitive events in the composite events are detected.
- The `<not>` element can contain only one `<object>` element that represents a primitive event.
- The `<notany>` element can be used in place of the `<not>` element to support a notion of disjunction within the negation rule.
- At the time of activation, the `by` attribute of the `<not>` element is executed to compute the deadline for the primitive events in the `<not>` element. The value for the `by` attribute can be expressed using the (database) `SYSTIMESTAMP` (to be set to the time of activation) or any date attribute in the other primitive events (including the event creation time attributes discussed in [Section 4.2](#)), or both. The SQL datetime function `SYSTIMESTAMP` returns the system date including fractional seconds and time zone of the system on which the database resides. So, the rule condition in the preceding example can also be expressed as follows:

```

<condition>
  <and equal="order.orderId, ship.orderId">
    <object name="order"> Type = 'Gold' </object>
    <not by="order.rlm$CrtTime+1">
      <object name="ship"/>
    </not>
  </and>
</condition>

```

Another variant of the preceding rule is one that uses a user-supplied date in the deadline computation. For example, a `ShipBy` attribute in the `PlaceOrder` event can hold the time by which the shipment is expected and the deadline can be computed using this attribute, such as shown here:

```

<condition>
  <and equal="order.orderId, ship.orderId">
    <object name="order"> Type = 'Gold' </object>
    <not by="order.ShipBy-1">
      <object name="ship"/>
    </not>
  </and>
</condition>

```

Rules with negation involving a deadline other than `SYSTIMESTAMP` are not allowed in a rule class with the `AUTOCOMMIT` property turned off (see [Section 3.7](#)). This also includes the rule classes configured for `DMLEVENTS` (see [Section 3.8](#)).

Rules involving negation constructs can be used to raise alerts (in corresponding rule actions) when a set of primitive events are generated out of order. In applications such as *Workflow*, rules are often used to enforce sequencing among various business events. The action of such rules is to raise an exception (alert an agent) when the

events are detected out of order. A `<not>` element without a hard deadline (no `by` attribute) can be used to define such rules.

Consider a composite event with three primitive events: `PlaceOrder`, `PaymentReceived`, and `ShipOrder`. A rule can be used to alert an agent (action) if the `ShipOrder` event is generated before the `PaymentReceived` event is detected. (Note that there are alternate ways to model this application in a Workflow system, but this approach is used to explain the negation concept). For this example, the composite event structure and the rule condition are represented as follows:

```
CREATE or REPLACE TYPE OrderTrack AS OBJECT (
    order PlaceOrder, -- primitive event type --
    pay PaymentReceived, -- primitive event type --
    ship ShipOrder); -- primitive event type --

<condition>
  <and equal="order.OrderId, pay.OrderId, ship.OrderId">
    <object name="order"/> -- no conditions on the primitive events --
    <object name="ship"/>
    <not>
      <object name="pay"/>
    </not>
  </and>
</condition>
```

The previous example uses a `<not>` element with no deadline specification (`by` attribute) and thus this value defaults to `SYSTIMESTAMP` (the time at which all other primitive events in the rule condition are detected). The `sequence="yes"` (Section 4.2) property, such as shown in the following example, can be used to ensure ordering among the detected events.

```
<condition>
  <and equal="order.OrderId, pay.OrderId, ship.OrderId" sequence="yes">
    <object name="order"/> -- no conditions on the primitive events --
    <object name="ship"/>
    <not>
      <object name="pay"/>
    </not>
  </and>
</condition>
```

In the previous rule condition, the deadline for the `PaymentReceived` event is determined by the occurrence of the `ShipOrder` event, which follows the corresponding `PlaceOrder` event. In effect, the action associated with the preceding rule condition will be executed if the `ShipOrder` event is detected before the `PaymentReceived` event for a particular order.

The negation construct can often be used to detect the non-occurrence of two or more primitive events. For example, a rule such as "If an order is placed by a Gold customer and the items are not shipped within 24 hours of the order placement or if the order is not cancelled, alert the representative" uses negation on the two events, `ShipOrder` and `CancelOrder`. Such rule conditions can be expressed using a `<notany>` element in the place of the `<not>` element as shown in the following example:

```
<condition>
  <and equal="order.orderId, ship.orderId, cancel.orderId">
    <object name="order"> Type = 'Gold' </object>
    <notany count=1 by="order.rlm$CrtTime+1">
      <object name="ship"/>
      <object name="cancel"/> -- assuming a CancelOrder event --
    </notany>
  </and>
</condition>
```

```

    </notany>
  </and>
</condition>

```

The primitive events appearing within the `<not>` or `<notany>` element should not be referenced in the `join` attribute specification of the `<and>` element. However, they (primitive events) can be used within the `EQUAL` property specifications. If there is a need to specify a join condition (other than those already captured by the `EQUAL` property specifications), the `join` attribute for the `<not>` element can be used. The conditional expression specified for this `join` attribute can reference all the primitive events that appear in the rule condition, including those appearing within the `<not>` element, such as shown in the following example:

```

<condition>
  <and equal="order.orderId, ship.orderId">
    <object name="order"> Type = 'Gold' </object>
    <not by="order.rlm$CrtTime+1"
      join="order.Address_zipcode = ship.Address_zipcode">
      <object name="ship"/>
    </not>
  </and>
</condition>

```

The rule condition with a negation is considered true only if the join condition in the `<and>` element evaluates to true and the join condition in the not condition evaluates to false (or there is no event that matches this criteria within specified deadline).

4.4 Rule Conditions with Set Semantics

In some applications, the primitive events that constitute a composite event can be the same structure. For example, `AddItem` could be a primitive event that is generated when a customer adds an item to his shopping cart. Rules can be defined to monitor multiple items added to the shopping cart and suggest new items based on the past customer experiences (association rules generated by a data mining tools).

Consider an electronics Web store that sells accessories for camcorders. A typical rule in their application could be "If a customer adds a camcorder lens worth more than \$100, a lens filter, and a IR light to the shopping cart, suggest a tripod to him". This rule consists of three simple conditions to be checked on every `AddItem` event generated in the system, such as shown in the following example:

```

Accessory = 'Lens' and Price > 100
Accessory = 'Lens Filter'
Accessory = 'IR Light'

```

To support the application described previously, the composite event structure can be modeled as an object type with multiple embedded types of the same primitive event type (`AddItem`) as shown in the example that follows. If required, the same composite event structure may also include other primitive event types.

```

CREATE or REPLACE TYPE AddItem AS OBJECT (
  Accessory VARCHAR(30),
  Make VARCHAR(20),
  Price NUMBER);
CREATE or REPLACE TYPE CrossSellEvent AS OBJECT (
  Item1 AddItem,
  Item2 AddItem,
  Item3 AddItem,
  Item4 AddItem,

```

```
Item5 AddItem);
```

The preceding composite event is created to accommodate rules that are monitoring at most five primitive events in the shopping cart. (Note that the shopping cart may still contain more than 5 items.) In this rule application, the events can be configured for SESSION duration (see [Section 3.4](#)) such that only the primitive events generated within a user session are considered for rule matches. Using the composite event rule condition syntax, the preceding condition can be expressed as follows:

```
<condition>
  <and>
    <object name="Item1"> Accessory = 'Lens' and Price > 100 </object>
    <object name="Item2"> Accessory = 'Lens Filter' </object>
    <object name="Item3"> Accessory = 'IR Light' </object>
  </and>
</condition>
```

Note that the element names `Item1`, `Item2`, and `Item3` are used to assign the matching events to appropriate attributes of the `CrossSellEvent` instance. Also, this assignment allows (join) predicates across primitive events in a rule condition as follows:

```
<condition>
  <and join="Item1.Price+Item2.Price+Item3.Price > 300">
    <object name="Item1"> Accessory = 'Lens' and Price > 100 </object>
    <object name="Item2"> Accessory = 'Lens Filter' </object>
    <object name="Item3"> Accessory = 'IR Light' </object>
  </and>
</condition>
```

4.5 Rule Conditions with Any n Semantics

The examples discussed so far use rules that match all the primitive events specified in a rule condition. This is achieved with the use of an `<and>` element as the parent of all the primitive event conditions. Some rule applications require rules that could match a subset of primitive events specified in the rule condition. For example, consider a composite event `CE1` consisting of three primitive events `PE1`, `PE2`, and `PE3`. Now, a rule condition defined for the composite event may need to match only one of the three primitive events. For this example, the composite event structure and the rule condition are represented as follows:

```
-- Composite event structure --
CREATE or REPLACE TYPE CE1 AS OBJECT (
    pe1Inst PE1,
    pe2Inst PE2,
    pe3Inst PE3);
-- Sample Rule condition --
<condition>
  <any>
    <object name="pe1Inst"/>
    <object name="pe2Inst"/>
    <object name="pe3Inst"/>
  </any>
</condition>
```

When the rule condition should match any two of the three primitive events, the `count` attribute of the `<any>` element can be used, as shown in the example that

follows. By default, the `count` attribute has a value of 1, which is equivalent to a disjunction (OR) of all the primitive events specified within the `<any>` element.

```
<condition>
  <any count=2>
    <object name="pe1Inst"/>
    <object name="pe2Inst"/>
    <object name="pe3Inst"/>
  </any>
</condition>
```

The Any *n* semantics in the rule conditions are very common in applications using set semantics. The rule considered in the cross-selling application of [Section 4.4](#) can be extended to suggest the tripod to the customer if the shopping cart has any two of the three items specified. The condition for this rule can be represented using the Any *n* syntax as follows:

```
<condition>
  <any count=2>
    <object name="Item1"> Accessory = 'Lens' and Price > 100 </object>
    <object name="Item2"> Accessory = 'Lens Filter' </object>
    <object name="Item3"> Accessory = 'IR Light' </object>
  </any>
</condition>
```

In a rule condition, some of the primitive events specified within an `<any>` list may be mandatory for the condition to evaluate to true. For example, in the preceding rule condition, the Lens (Item1) may be mandatory and it should always count for one item in two items matched with the `<any count=2>` specification. This new rule condition can be represented using the join attribute of the `<any>` element as follows:

```
<condition>
  <any count=2 join="Item1 IS NOT NULL">
    <object name="Item1"> Accessory = 'Lens' and Price > 100 </object>
    <object name="Item2"> Accessory = 'Lens Filter' </object>
    <object name="Item3"> Accessory = 'IR Light' </object>
  </any>
</condition>
```

Within an `<any>` list, often there is a need to correlate the primitive events that occur. For example, the preceding rule can be extended to suggest the tripod to the customer only if the `Make` attribute of the two items matched is same. When using an `<and>` element (to match all three items), this can be posed as a join predicate on the `Make` attribute of each primitive event, such as shown in the following example:

```
<condition>
  <and join="Item1.Make=Item2.Make and Item2.Make=Item3.Make">
    <object name="Item1"> Accessory = 'Lens' and Price > 100 </object>
    <object name="Item2"> Accessory = 'Lens Filter' </object>
    <object name="Item3"> Accessory = 'IR Light' </object>
  </and>
</condition>
```

However, similar join predicates cannot be used to correlate primitive events in an `<any>` list because the missing primitive events (the one left out in 2 out of 3) are represented as NULLs and any predicate (other than IS NULL) on a NULL value is always false. For this purpose, when using the `<any count=2>` specification, the rule should use the following join condition:

```
(Item1.Make is null and Item2.Make = Item3.Make) or
(Item2.Make is null and Item1.Make = Item3.Make) or
```

```
(Item3.Make is null and Item1.Make = Item2.Make)
```

Within an `<any>` element, the preceding join condition can be represented in an abbreviated form using an equal clause. With this syntax, the join condition works well with any value assigned to the count attribute of the `<any>` element, such as shown in the following example:

```
<condition>
  <any count=2 equal="Item1.Make, Item2.Make, Item3.Make">
    <object name="Item1"> Accessory = 'Lens' and Price > 100 </object>
    <object name="Item2"> Accessory = 'Lens Filter' </object>
    <object name="Item3"> Accessory = 'IR Light' </object>
  </any>
</condition>
```

The equality joins among primitive events of a composite event are very common and thus this abbreviated syntax is supported for `<and>` element as well, as shown in the following example:

```
<condition>
  <and equal="Item1.Make, Item2.Make, Item3.Make">
    <object name="Item1"> Accessory = 'Lens' and Price > 100 </object>
    <object name="Item2"> Accessory = 'Lens Filter' </object>
    <object name="Item3"> Accessory = 'IR Light' </object>
  </and>
</condition>
```

When both `equal` and `join` attributes are used in an `<and>` or an `<any>` element, the join predicates represented by the equal specification are combined (using logical AND) with the join predicates listed with the `join` attribute. For example, the following condition matches any two specified items which are of same make and whose total value is greater than 300. (Note the use of NVL functions in the join predicates).

```
<condition>
  <any count=2 equal="Item1.Make, Item2.Make, Item3.Make"
    join="NVL(Item1.Price,0) + NVL(Item2.Price,0) + NVL(Item3.Price,0) > 300">
    <object name="Item1"> Accessory = 'Lens' and Price > 100 </object>
    <object name="Item2"> Accessory = 'Lens Filter' </object>
    <object name="Item3"> Accessory = 'IR Light' </object>
  </any>
</condition>
```

The use of equal attribute at the rule class level (instead of each rule) is discussed in [Section 3.5](#).

The `sequence` attribute ([Section 4.2](#)) can be used in an `<any>` element to ensure that the matching primitive events happen in the specified order for the rule condition to evaluate to true.

```
<condition>
  <any count=2 sequence="yes">
    <object name="Item1"> Accessory = 'Lens' and Price > 100 </object>
    <object name="Item2"> Accessory = 'Lens Filter' </object>
    <object name="Item3"> Accessory = 'IR Light' </object>
  </any>
</condition>
```

Rules Applications That Span Multiple Tiers

Rules applications can be run in multitier mode. For rules applications that span multiple tiers where rule management is handled in the database, but the action execution for the rules is handled in the application server, the actions for the rules matching an event cannot be invoked from an action callback procedure. Instead, a results view is populated with the information about the events and matching rules; both of which are available for external action execution. The results view can be queried to determine the rules that match an event and their corresponding actions can then be executed.

5.1 Creating Rules Applications That Span Multiple Tiers

To handle rules applications with certain rules having their action execution occurring on the application server, you must configure the rule class for external execution (in addition to configuring the action callback procedure). The steps to do this are similar to those described in [Section 2.3](#), but are modified as follows:

1. Create the event structure as an object type in the database (same as Step 1 in [Section 2.3](#)).
2. Create the `TravelPromotion` rule class. Also define the results view, even though you may not use it initially. The results view can be used, for example, to create the `TravelPromotion` rule class, such that for each rule session (that processes some events for a rule class), the action execution can switch at runtime between either the action callback procedure (calling `dbms_rlmgr.process_rules()` as shown in Step 5 in [Section 2.3](#)) or external action execution (calling `dbms_rlmgr.add_event()`, as shown in Step 5 in this section). For this purpose, the rule class is configured with an action callback procedure and a results view, as shown in the following example:

```
BEGIN
  dbms_rlmgr.create_rule_class (
    rule_class    => 'TravelPromotion',
    event_struct => 'AddFlight',
    action_cbk    => 'PromoAction',
    rslt_viewnm   => 'MatchingPromos',
    actprf_spec  => 'PromoType VARCHAR2 (20),
                  OfferedBy VARCHAR2 (20)');
END;
```

Note that this command creates the following `MatchingPromos` results view to hold the results from the rule evaluation. This view has a fixed set of columns to list the system generated event identifier (`rlm$eventid`), the event instance (`rlm$event` for a (simple) primitive event), the rule identifier for the matching rules (`rlm$ruleid`), the rule condition (`rlm$rulecond`), the rule description

(`rlm$ruledesc`), and a variable set of columns to represent the action preferences associated with the rules (`PromoType` and `OfferedBy` columns in this example). For an event matching a set of rules in the rule class, the information about the event and the matched rules can be obtained by querying this view.

```
VIEW MatchingPromos (
    rlm$eventid    ROWID,
    rlm$event      AddFlight,
    rlm$ruleid     VARCHAR2(100),
    PromoType      VARCHAR2(20),
    OfferedBy      VARCHAR2(20),
    rlm$rulecond   VARCHAR2(4000),
    rlm$ruledesc   VARCHAR2(1000);
);
```

The results view in the case of a rule class configured for a composite event is structured to hold the results from evaluating the rules using one or more primitive events. For this purpose, this view is created with separate columns for each primitive event within the composite event. For example, the following results view is created for the rule class defined in [Section 2.4.1](#):

```
VIEW CompMatchingPromos (
    rlm$eventid    ROWID,
    Flt            AddFlight,
    Car            AddRentalCar,
    rlm$ruleid     VARCHAR2(100),
    PromoType      VARCHAR2(20),
    OfferedBy      VARCHAR2(20),
    rlm$rulecond   VARCHAR2(4000),
    rlm$ruledesc   VARCHAR2(1000));
```

3. Implement the action callback procedure (same as Step 3 in [Section 2.3](#)).
4. Add rules to the rule class (same as Step 4 in [Section 2.3](#)).
5. Identify the matching rules for an event. This step replaces the use of the process rules procedure (`dbms_rlmgr.process_rules()`) that identifies the matching rules and executes the corresponding actions with an add event procedure (`dbms_rlmgr.add_event()`) that adds the event to the rule class one at a time and identifies the matching rules for a given event that are later accessed using the `MatchingPromos` results view.

```
BEGIN
dbms_rlmgr.add_event (
    rule_class => 'TravelPromotion',
    event_inst => AddFlight.getVarchar(987, 'Abcair', 'Boston', 'Orlando',
    '01-APR-2003', '08-APR-2003'));
END;
```

6. Find the matching rules by querying the results view. For example, the following query returns a list of all the events added in the current session and their corresponding matching rules (and their action preferences):

```
SELECT rlm$eventid, rlm$ruleid, PromoType, OfferedBy FROM MatchingPromos;
```

The results from this query can be used to execute the appropriate action in the application server. In the case of a rule class defined for a single event structure, this view implicitly has a `rlm$eventid` column that returns a system generated event identifier and `rlm$event` column to return the actual event as the (primitive event structure's) object instance.

When you need to identify one candidate rule from the result set (conflict resolution), you can use `ORDER BY`, `GROUP BY`, and `HAVING` clauses. Note that the callback mechanism for action execution can only use `ORDER BY` semantics for conflict resolution. See [Section 3.3](#) for more information. For example, if the Travel Services application offers only one promotion of each type, the following analytical query can be used to identify the appropriate rules to be fired:

```
SELECT rlm$eventid, rlm$ruleid, PromoType, OfferedBy
FROM (SELECT rlm$eventid, rlm$ruleid, PromoType, OfferedBy,
            ROW_NUMBER( ) over (PARTITION BY PromoType
                                ORDER BY rlm$ruleid) rnum
      FROM MatchingPromos)
WHERE rnum=1;
```

In this example, the rule identified as the one to be fired is the first one (`rnum=1`) returned from the query of the result set for the set of rules that evaluated to be true, partitioned by the type of promotion and ordered in ascending order by the `rlm$ruleid` column value.

The results from a rule evaluation are available until the end of the rule session. By default, the database session (from connect to disconnect) is considered the rule session. Alternatively, the reset session procedure (`dbms_rlmgr.reset_session()`) can be used to end a rule session and start a new session within a database session. Note that at the beginning of a rule session, the results view is empty.

7. Consume the event that is used in a rule execution. An event can be marked for exclusive or shared execution of rules by specifying the consumption policy for the events. Previously, in [Section 2.3](#), if the `TravelPromotion` rule class was configured for exclusive consumption of events, then an event used for the execution of a rule was immediately deleted from the system and it could not be used for any other (matching) rules. Because the action callback procedure is used, the rule manager automatically handles the consumption of the exclusive events. However, when external action execution is used, the application should explicitly consume the event chosen for an action execution by using the consume event procedure (`dbms_rlmgr.consume_event()`). This procedure ensures that when multiple concurrent sessions try to consume the same event, only one of them succeeds. So, the action for a particular rule should be executed if the event is successfully consumed, as follows:

```
DECLARE
    consumed number;
BEGIN
    consumed := dbms_rlmgr.consume_event (
                rule_class => 'TravelPromotion',
                event_ident => :eventIdBind);
    IF (consumed = 1) THEN
        OfferPromotion(...); -- offer the promotion only if the event
                             -- consumption is successful
    END IF;
END;
```

The event identifier is obtained from the value listed in the `rlm$eventid` column of the `MatchingPromos` results view. If the consumption policy (see [Section 3.3](#)) for all events is shared, then the `CONSUME_EVENT` call always returns 1 and the event is still available. Note that only the events configured for exclusive consumption are consumed and the corresponding rows from the results view are deleted.

5.2 Modes of Operation

A rule-based application designed using Rules Manager has a varying number of steps, depending on its mode of operation. Almost all the steps in both cases, single tier and multitier, are one-time implementations. Once these implementations are in place, the end-user no longer needs to deal with the Rule Manager APIs. The new rules are added using the SQL `INSERT` statement against the rule database table and the run-time calls that are embedded in larger applications will automatically process these new rules.

A rule class stored in the database can operate in either of the following two modes:

- **Single tier mode** -- the rule evaluation, identification of the candidate rules or action for execution, execution of action, and optional consumption of events all happen in the database with a single `PROCESS_RULES` call (which passes in the event instance). Note that this is the most common case even for applications running outside the database.
- **Multitier mode** -- the rule evaluation happens in the database and the remaining steps described in single-tier mode can be done in any tier with appropriate database calls (with a maximum of four steps, which are described in [Section 5.2.2](#)).

5.2.1 Single Tier Mode

See either [Section 2.3](#) or [Section 2.4](#) for an example of a rule class stored in the database that uses a single tier mode of operation.

5.2.2 Multitier Mode

The main reasons for a rules application to operate in the multitier mode are:

- The action suggested by the rules cannot be implemented as a database function or package (PL/SQL or Java) in the database.
- The conflict resolution criterion for the rule class is complex and it cannot be specified using a SQL `ORDER BY` clause. In situations when a single event processing a set of rules matches two or more rules, conflict resolution criterion is used to identify a subset of rules or determine an exact order of rules that should be fired, or both. Using a simple SQL `ORDER BY` clause is usually sufficient for most applications. However, multitier mode can make use of any SQL operator (including analytical operators) for the conflict resolution criterion.

The four steps to use Rules Manager in the multitier mode are:

1. Tell the database about the event by calling the `dbms_rlmgr.add_event` procedure.
2. Ask the database which rules apply (query a view, possibly with a complex query with a SQL `ORDER BY` clause, and so forth).
3. Based on the applications conflict-resolution criteria, identify a subset of the matched rules that should be fired and prepare for executing the action by consuming the event with a `dbms_rlmgr.consume_event` function call.
4. Upon success in Step 3, make calls to the (local, middle tier resident) routines that the programmer maps to the actions that are defined.

If the only reason for using the multitier mode is to execute the actions in the application server, then the single tier mode with a few modifications can be used (thereby reducing the number of steps involved to two). The action callback procedure

in the single-tier mode can be implemented to enqueue the actions and continue with the rest of the operations (consumption). The application server can subscribe to this action queue and execute the actions. In this configuration a minimum of two database calls are required (`PROCESS_RULES` call and `DEQUEUE` call).

Rules Manager, as a database feature, can be used in multiuser and concurrent session environments. It allows two concurrent sessions to process the same set of rules and call for deletion of a common event that matched the rules and ensures that only one of the sessions succeeds. When the rule application is operating in the single-tier mode, this happens by specifying an `EXCLUSIVE` consumption policy for the event type. The `PROCESS_RULE` procedure controls the event consumption logic and avoids deadlocks between various sessions. When the rule application is operating in multitier mode, the middle tier application must signal its intent to execute the action of a rule by calling the `CONSUME_EVENT` function (because the user application is controlling the conflict resolution criterion). This call returns 0 if any one of the events required by the action has already been consumed by another concurrent session. So, the application should execute the action only if this call returns 1. Note that this step can be skipped if all the events are configured for `SHARED` consumption (implying that the events are shared for multiple rule executions).

Because one of the main reasons for using the multitier mode is to implement complex conflict resolution criteria, the results from matching an event with the rules is exposed (to the application) as a relation that can be queried using complex SQL. This view can also be used to specify different resolution criteria based on some external factors (for example, use one conflict-resolution criterion between the times 9AM-5PM and other criterion for the rest of the day).

5.2.2.1 Actions in the Mid-Tier

Rules Manager rule classes can store any form of data (scalar, XML, Raw, BLOB, and so forth) along with the rule definition. This data is returned back to the action-callback procedure or the application when the corresponding rule matches an event.

For example, a rule application may choose to store Simple Object Access Protocol (SOAP) messages in their full form (in an `XMLType` column) as actions for each rule. So, when a rule matches an event, this SOAP message is returned to the application. The application in the middle tier could interpret the data accordingly and perform the required action (post the SOAP message). See [Appendix G](#) for additional information on action execution.

In another application, the exact call for the action may be fixed, for example using the `OfferDiscount2Customer` function. In this case, the rule definitions may just store the percentage of discount that should be offered. When this discount value is returned to the application, it can be bound as an argument to the `OfferDiscount2Customer` function call.

Rules Manager Object Types

Rules Manager is supplied with one predefined type and a public synonym for this type. [Table 6-1](#) describes the Rules Manager object type.

Tip: See the "Rules Manager Types" chapter in *Oracle Database PL/SQL Packages and Types Reference* for all reference information concerning Rules Manager object types.

Table 6-1 Rules Manager Object Types

Object Type Name	Description
RLM\$EVENTIDS	Specifies a list of event identifiers to the CONSUME_PRIM_EVENTS procedure

DBMS_RLMGR Package

Rules Manager uses the DBMS_RLMGR package, which contains various procedures, to create and manage rules and rule sessions. The following table describes the procedures in the DBMS_RLMGR package.

None of the values and names passed to the procedures defined in the DBMS_RLMGR package are case insensitive, unless otherwise mentioned. In order to preserve the case, double quotation marks should be used around the values.

Tip: See the "DBMS_RLMGR" chapter in *Oracle Database PL/SQL Packages and Types Reference* for all reference information concerning Rules Manager package procedures.

Table 7-1 DBMS_RLMGR Procedures

Procedure	Description
ADD_ELEMENTARY_ATTRIBUTE procedure	Adds the specified attribute to the event structure (and the Expression Filter attribute set)
ADD_EVENT procedure	Adds an event to a rule class in an active session
ADD_FUNCTIONS procedure	Adds a function, a type, or a package to the approved list of functions with an event structure (also the Expression Filter attribute set)
ADD_RULE procedure	Adds a rule to the rule class
CONSUME_EVENT function	Consumes an event using its identifiers and prepares the corresponding rule for action execution
CONSUME_PRIM_EVENTS function	Consumes one or more primitive events with all or none semantics
CREATE_EVENT_STRUCTURE procedure	Creates an event structure
CREATE_RULE_CLASS procedure	Creates a rule class
DELETE_RULE procedure	Deletes a rule from a rule class
DROP_EVENT_STRUCTURE procedure	Drops an event structure
DROP_RULE_CLASS procedure	Drops a rule class
GRANT_PRIVILEGE procedure	Grants a privilege on a rule class to another user
PROCESS_RULES procedure	Processes the rules for a given event
RESET_SESSION procedure	Starts a new rule session within a database session
REVOKE_PRIVILEGE Procedure	Revokes a privilege on a rule class from a user

Rules Manager Views

Rules Manager metadata can be viewed using the Rules Manager views defined with a xxx_RLMGR prefix, where xxx can be the string USER or ALL. These views are read-only to users and are created and maintained by the Rules Manager procedures.

Table 8–1 lists the names of the views and their descriptions.

Table 8–1 Rules Manager Views

View Name	Description
USER_RLMGR_EVENT_STRUCTS View	List of all event structures in the current schema
USER_RLMGR_RULE_CLASSES View	List of all rule classes in the current schema
USER_RLMGR_RULE_CLASS_STATUS View	List of the progress of rule class creation
USER_RLMGR_PRIVILEGES View	List of the privileges for the rule class
USER_RLMGR_COMPRCLS_PROPERTIES View	List of primitive events configured for a rule class and the properties for each event

8.1 USER_RLMGR_EVENT_STRUCTS View

The USER_RLMGR_EVENT_STRUCTS view lists all the event structures in the current schema. This view is defined with the columns listed and described in Table 8–2.

Table 8–2 USER_RLMGR_EVENT_STRUCTS View

Column Name	Data Type	Description
EVENT_STRUCTURE_NAME	VARCHAR2	Specifies the name of the event structure
HAS_TIMESTAMP	VARCHAR2	Specifies whether the event structure has the event creation timestamp - YES/NO
IS_PRIMITIVE	VARCHAR2	Specifies whether the event structure is strictly primitive - YES/NO

8.2 USER_RLMGR_RULE_CLASSES View

The USER_RLMGR_RULE_CLASSES view lists all the rule classes in the current schema. This view is defined with the columns listed and described in Table 8–3.

Table 8–3 USER_RLMGR_RULE_CLASS View

Column Name	Data Type	Description
RULE_CLASS_NAME	VARCHAR2	Name of the rule class

Table 8–3 (Cont.) USER_RLMGR_RULE_CLASS View

Column Name	Data Type	Description
ACTION_CALLBACK	VARCHAR2	The procedure configured as the action callback for the rule class
EVENT_STRUCTURE	VARCHAR2	The event structure used for the rule class
RULE_CLASS_PACK	VARCHAR2	Name of the package implementing the rule class cursors (internal)
RCLS_RSLT_VIEW	VARCHAR2	View to display the matching events and rules for the current session
IS_COMPOSITE	VARCHAR2	Indicates whether the rules class is configured for composite events; if so, the value is YES
SEQUENCE_ENB	VARCHAR2	Indicates whether the rules class is enabled for rule conditions with sequencing; if so, the value is YES
AUTOCOMMIT	VARCHAR2	Indicates whether the rules class is configured for auto-committing events and rules; if so, the value is YES
CONSUMPTION	VARCHAR2	Default Consumption policy for the events in the rule class: valid values are EXCLUSIVE and SHARED
DURATION	VARCHAR2	Default Duration policy of the primitive events
ORDERING	VARCHAR2	Ordering clause used for conflict resolution among matching rules and events
EQUAL	VARCHAR2	Equal specification for the rule classes configured for composite events
DML_EVENTS	VARCHAR2	Types of DML operations enabled for event management

8.3 USER_RLMGR_RULE_CLASS_STATUS View

The USER_RLMGR_RULE_CLASS_STATUS view lists the progress of rule class creation. This view is defined with the columns listed and described in [Table 8–4](#).

Table 8–4 USER_RLMGR_RULE_CLASS_STATUS View

Column Name	Data Type	Description
RULE_CLASS_NAME	VARCHAR2	Name of the rule class
STATUS	VARCHAR2	Current status of the rule class
STATUS_CODE	VARCHAR2	Internal code for the status
NEXT_OPERATION	VARCHAR2	Next operation performed on the rule class

8.4 USER_RLMGR_PRIVILEGES View

The USER_RLMGR_PRIVILEGES view lists the privileges for the rule classes. This view is defined with the columns listed and described in [Table 8–5](#).

Table 8–5 USER_RLMGR_PRIVILEGES View

Column Name	Data Type	Description
RULE_CLASS_OWNER	VARCHAR2	Owner of the rule class
RULE_CLASS_NAME	VARCHAR2	Name of the rule class

Table 8–5 (Cont.) USER_RLMGR_PRIVILEGES View

Column Name	Data Type	Description
GRANTEE	VARCHAR2	Grantee of the privilege. Current user or PUBLIC
PRCS_RULE_PRIV	VARCHAR2	Current user's privilege to execute or process rules
ADD_RULE_PRIV	VARCHAR2	Current user's privilege to add new rules to the rule class
DEL_RULE_PRIV	VARCHAR2	Current user's privilege to delete rules

8.5 USER_RLMGR_COMPRCLS_PROPERTIES View

The USER_RLMGR_COMPRCLS_PROPERTIES view lists the primitive events configured for a rule class and their properties. This view is defined with the columns listed and described in [Table 8–6](#).

Table 8–6 USER_RLMGR_COMPRCLS_PROPERTIES View

Column Name	Data Type	Description
RULE_CLASS_NAME	VARCHAR2	Name of the rule class configured for composite rules
PRIM_EVENT	VARCHAR2	Name of the primitive event in the composite event
PRIM_EVENT_STRUCT	VARCHAR2	Name of the primitive event structure (object type)
HAS_CRTTIME_ATTR	VARCHAR2	Whether the primitive event structure has the RLM\$CRTTIME attribute; if so, the value is YES
CONSUMPTION	VARCHAR2	Consumption policy for the primitive event: valid values are EXCLUSIVE and SHARED
TABLE_ALIAS_OF	VARCHAR2	Table name for the table alias primitive event
DURATION	VARCHAR2	Duration policy for the primitive event

Rules Manager Use Cases

This chapter describes a Law Enforcement application and an Order Management application to demonstrate the use of Rules Manager in multiple configurations and to demonstrate the expressiveness of the complex rule conditions.

Note: The complete scripts for these two applications can be found installed at: `$ORACLE_HOME/rdbms/demo` as `ruldemo.sql`.

9.1 Law Enforcement Rules Application

In this application, rules are defined to raise security alerts, place a person on the watch list, and so forth based on certain criteria. For this purpose, some real-world events such as bank transactions, transportation, and field reports are used to describe the criteria.

The basic steps to create the Law Enforcement rules application with composite events are as follows:

1. Create the table `messagequeue` to hold the messages with a timestamp value:

```
create table messagequeue (attime timestamp, mesg varchar2(4000));
```

2. Create the basic types that represent the event structure:

```
create or replace type BankTransaction as object
(subjectId NUMBER,          --- Refer to entity such as personnel
                                --- Could be SSN and so forth
 tranType  VARCHAR2(30),    --- DEPOSIT / TRANSFER / WITHDRAW
 amount    NUMBER,         ---
 fundFrom  VARCHAR2(30));  --- Location from which it is transferred
/
```

```
create or replace type Transportation as object
(subjectId  NUMBER,
 vesselType VARCHAR2(30),  --- TRUCK / CAR / PLANE / TRAIN
 locFrom    VARCHAR2(30),  --- Starting location
 locTo      VARCHAR2(30),  --- Ending location
 startDate  DATE,         --- start date
 endDate    DATE);       --- end date
/
```

```
create or replace type FieldReport as object
(subjectId NUMBER,
 rptType   VARCHAR2(30),  --- Tel call / Meeting / Bg Check
 whoWith   NUMBER,       --- Identifier of the person with whom
                                --- the subject is in touch
/
```

```

    rptOrg    VARCHAR2(30),    --- Organization reporting it
    rptReg    VARCHAR2(30),    --- Region
    rptBody   sys.XMLType);    --- The actual report
/

```

3. Create a composite event type that consists of the basic types defined in Step 2:

```

create or replace type LawEnforcement as object
    (bank      BankTransaction,
     transport Transportation,
     fldrpt    FieldReport);
/

```

4. Create a database table for the rules defined on the composite event structure:

```

BEGIN
    DBMS_RLMGR.CREATE_RULE_CLASS (
        rule_class    => 'LawEnforcementRC',
        event_struct  => 'LawEnforcement',
        action_cbk    => 'LawEnforcementCBK',
        actprf_spec   => 'actionType VARCHAR2(40), actionParam VARCHAR2(100)',
        rslt_viewmm   => 'MatchedCriteria',
        rlcls_prop    => '<composite
                        equal="bank.subjectId, transport.subjectId, fldrpt.subjectId"
                        ordering="rlm$rule.rlm$ruleid, bank.subjectId, transport.subjectId"/>');
END;
/

```

The rule class `LawEnforcementRC` is a relational table that acts as the repository for rules. This table has a set of predefined columns to store the rule identifiers, rule conditions and the descriptions. In addition to these columns, this rule class table is defined with two columns, `actionType` and `actionParam`, as specified through the `actprf_spec` argument. These columns capture the type of action that should be carried for each rule. For example:

```

desc LawEnforcementRC;

```

Name	Null?	Type
RLM\$RULEID		VARCHAR2(100)
ACTIONTYPE		VARCHAR2(40)
ACTIONPARAM		VARCHAR2(100)
RLM\$RULECOND		VARCHAR2(4000)
RLM\$RULEDESC		VARCHAR2(1000)

This step also creates the skeleton for an action callback procedure with the specified name. For example:

```

select text from user_source where name = 'LAWENFORCEMENTCBK' order by line;

```

```

TEXT
-----
procedure "LAWENFORCEMENTCBK" ( "BANK" "BANKTRANSACTION",
    "TRANSPORT" "TRANSPORTATION",
    "FLDRPT" "FIELDREPORT",
    rlm$rule "LAWENFORCEMENTRC"%ROWTYPE) is
begin
    null;
    --- The action for the matching rules can be carried here.
    --- The appropriate action can be determined from the
    --- event and action preferences associated with each rule.
end;

```

10 rows selected.

5. Implement the callback procedure to perform the appropriate action for each matching rule, based on the event instances that matched the rule and the action preferences associated with the rule. For this use case, a detailed message inserted into the message queue table is considered the action for the rules. For example:

```
CREATE OR REPLACE PROCEDURE LAWENFORCEMENTCBK (
    bank          banktransaction,
    transport     transportation,
    fldrpt        fieldreport,
    rlm$rule      LawEnforcementRC%ROWTYPE) IS
    msg           VARCHAR2(4000);
    msg1          VARCHAR2(100);
begin
    msg1 := 'Rule '||rlm$rule.rlm$ruleid||' matched following primitive events';
    dbms_output.put_line(msg1);
    msg := msg1||chr(10);
    if (bank is not null) then
        msg1 := '->Bank Transaction by subject ('||bank.subjectId||') of type
                ['||bank.tranType||'];

        dbms_output.put_line(msg1);
        msg := msg||msg1||chr(10);
    end if;
    if (transport is not null) then
        msg1 :=
            '->Transportation by subject('||transport.subjectId||') use vessel
                ['||transport.vesselType||'];

        dbms_output.put_line(msg1);
        msg := msg||msg1||chr(10);
    end if;
    if (fldrpt is not null) then
        msg1 :=
            '->Field report refer to('||fldrpt.subjectId||' and '||fldrpt.whowith||');
        dbms_output.put_line(msg1);
        msg := msg||msg1||chr(10);
    end if;

    msg1 := '>Recommended Action : Action Type ['||rlm$rule.actionType||
            ' ] Action Parameter ['||rlm$rule.actionParam||'];
    dbms_output.put_line(msg1||chr(10));
    msg := msg||msg1||chr(10);
    insert into messagequeue values (systimestamp, msg);
end;
/
```

6. The rules defined in the rules class can make use of user-defined functions in the database schema. The commands in the following list create some dummy functions that are later used in the rule conditions.

- a. For the value of the region passed in, query the restricted areas table and return 1 if the current region is a restricted area:

```
CREATE OR REPLACE FUNCTION IsRestrictedArea(region VARCHAR2)
    RETURN NUMBER IS
BEGIN
    -- User can expand this function and implement a logic
    -- that relies on other relational tables.
    RETURN 1;
END;
```

/

- b.** Check to see if the subject chosen is on the watch list and return 1 if True:

```
CREATE OR REPLACE FUNCTION OnWatchList(subject NUMBER)
                                RETURN NUMBER IS
BEGIN
-- User can expand this function and implement a logic
-- that relies on other relational tables.
RETURN 1;
END;
/
```

- c.** Check to see if the two parties are associates and return 1 if the two subjects passed in are associates according to the registry:

```
CREATE OR REPLACE FUNCTION AreAssociates(subjectA NUMBER,
                                         subjectB NUMBER)
                                RETURN NUMBER IS
BEGIN
-- User can expand this function and implement a logic
-- that relies on other relational tables.
RETURN 1;
END;
/
```

- d.** Add all three user-defined functions to the composite event LawEnforcement:

```
EXEC DBMS_RLMGR.ADD_FUNCTIONS('LawEnforcement', 'OnWatchList');
EXEC DBMS_RLMGR.ADD_FUNCTIONS('LawEnforcement', 'IsRestrictedArea');
EXEC DBMS_RLMGR.ADD_FUNCTIONS('LawEnforcement', 'AreAssociates');
```

- 7.** Define the rules that suggest some actions:

- a.** Rule: Add a person to the NYPD watch list if he receives a money transfer for more than \$10,000 and he rents a truck, one way, to one of the restricted areas. Note that the join predicate is specified at the rule class level.

```
INSERT INTO LawEnforcementRC (rlm$ruleid, actionType, actionParam,
                             rlm$rulecond)
VALUES ('1', 'ADD2WATCHLIST', 'NYPD',
        '<condition>
        <and>
        <object name="bank">
          tranType = 'TRANSFER' AND amount > 10000 AND
          fundFrom != 'usa'
        </object>
        <object name="transport">
          vesselType = 'TRUCK' AND locFrom != locTo AND
          IsRestrictedArea(locTo)=1
        </object>
        </and>
        </condition>');
```

- b.** Rule: Add a person to the NYPD watch list if two of the following three conditions are met. The person gets a money transfer for over \$10,000 from outside the United States, he rented a truck, one-way, into one of the restricted areas, and he had a phone conversation with a person already on the watch list. The following rule demonstrates the use of the <ANY> element where a rule condition is considered true if *m* out of *n* events are detected.

```
INSERT INTO LawEnforcementRC (rlm$ruleid, actionType, actionParam,
```

```

rlm$rulecond)
VALUES ('2', 'ADD2WATCHLIST', 'NYPD',
    '<condition>
    <any count="2">
    <object name="bank">
        tranType = ''TRANSFER'' AND amount > 10000 AND
        fundFrom != ''usa''
    </object>
    <object name="transport">
        vesselType = ''TRUCK'' AND locFrom != locTo AND
        IsRestrictedArea(locTo)=1
    </object>
    <object name="fldrpt">
        rptType = ''TELCALL'' AND OnWatchList(whoWith) = 1
    </object>
    </any>
</condition>');

```

- c. Rule: Start a background check on a person if he receives a large sum of money from outside the United States, he rents a truck one-way into one of the restricted areas, and there is no field report with his background information. The following rule demonstrates the use of negation where a rule condition is considered true if some of the specified events are detected and the other events are not detected:

```

INSERT INTO LawEnforcementRC (rlm$ruleid, actionType, actionParam,
rlm$rulecond)
VALUES ('3', 'STARTBACKGROUNDCHECK', 'RENTAL_DESTINATION',
    '<condition>
    <and>
    <object name="bank">
        tranType = ''TRANSFER'' AND amount > 10000 AND
        fundFrom != ''USA''
    </object>
    <object name="transport">
        vesselType='''TRUCK'' AND locFrom != locTo AND
        IsRestrictedArea(locTo)=1
    </object>
    <not>
        <object name="fldrpt"/>
    </not>
    </and>
</condition>');

```

- d. Rule: If a subject received over \$10,000 from outside the United States, he rented a truck for one way trip into a restricted area and a field report saying that the subject was never arrested before was not submitted within "certain" (0.001 fraction of a day; this could be days, but seconds are used to demonstrate the use of a deadline) period, add the destination of the truck to high-risk areas. This rule demonstrates Negation with a deadline:

```

INSERT INTO LawEnforcementRC (rlm$ruleid, actionType, actionParam,
rlm$rulecond)
VALUES ('4', 'ADD2HIGH_RISK_AREA', 'RENTAL_DESTINATION',
    '<condition>
    <and>
    <object name="bank">
        tranType = ''TRANSFER'' AND amount > 10000 AND
        fundFrom != ''usa''
    </object>

```

```

        <object name="transport">
            vesselType = 'TRUCK' AND locFrom != locTo AND
                IsRestrictedArea(locTo)=1
        </object>
        <not by="systimestamp+0.001">
            <object name="fldrpt">
                rptType = 'BACKGROUNDCHECK' and
                    extract(rptBody, '/history/arrests[@number=0']
                        is not null
            </object>
        </not>
    </and>
</condition>');

```

- e. Browse the rules. This is optional. The following example demonstrates this task:

```
select rlm$ruleid, rlm$rulecond from LawEnforcementRC order by 1;
```

```
RLM$R RLM$RULECOND
```

```

-----
1      <condition>
      <and>
        <object name="bank">
            tranType = 'TRANSFER' AND amount > 10000 AND fundFrom != 'usa'
        </object>
        <object name="transport">
            vesselType = 'TRUCK' AND locFrom != locTo AND
                IsRestrictedArea(locTo)=1
        </object>
      </and>
    </condition>
.
.
.

```

- 8. Process the rules for the primitive events.

- a. Add two primitive events that each partially match one or more rules and together match one rule, such that the rules action is executed (the message is printed to the screen as well as inserted into the messagequeue table):

```

set serveroutput on size 10000;
BEGIN
  dbms_rlmgr.process_rules (
    rule_class => 'LawEnforcementRC',
    event_inst =>
      sys.anydata.convertobject(
        fieldreport(123302122, 'TELCALL',123302123, 'NSA', 'NE', null)));
END;
/

BEGIN
  dbms_rlmgr.process_rules (
    rule_class => 'LawEnforcementRC',
    event_inst =>
      sys.anydata.convertobject(
        banktransaction(123302122, 'TRANSFER', 100000, 'USSR')));
END;
/
Rule 2 matched following primitive events

```

```

->Bank Transaction by subject (123302122) of type [TRANSFER]
->Field report refer to(123302122 and 123302123)
=>Recommended Action : Action Type [ADD2WATCHLIST] Action Parameter [NYPD]

```

- b.** The following Transportation event, in combination with the Bank Transaction event, evaluates some of the rules to true and thus calls the action call-back procedure with appropriate arguments:

```

BEGIN
  dbms_rlmgr.process_rules (
    rule_class => 'LawEnforcementRC',
    event_inst =>
      sys.anydata.convertobject(
        transportation(123302122, 'TRUCK', 'WIS', 'MD',
          sysdate, sysdate + 7));
END;
/
Rule 1 matched following primitive events
->Bank Transaction by subject (123302122) of type [TRANSFER]
->Transportation by subject(123302122) use vessel [TRUCK]
=>Recommended Action : Action Type [ADD2WATCHLIST] Action Parameter [NYPD]

Rule 2 matched following primitive events
->Transportation by subject(123302122) use vessel [TRUCK]
->Field report refer to(123302122 and 123302123)
=>Recommended Action : Action Type [ADD2WATCHLIST] Action Parameter [NYPD]

Rule 2 matched following primitive events
->Bank Transaction by subject (123302122) of type [TRANSFER]
->Transportation by subject(123302122) use vessel [TRUCK]
=>Recommended Action : Action Type [ADD2WATCHLIST] Action Parameter [NYPD]

Rule 3 matched following primitive events
->Bank Transaction by subject (123302122) of type [TRANSFER]
->Transportation by subject(123302122) use vessel [TRUCK]
=>Recommended Action : Action Type [STARTBACKGROUNDCHECK] Action Parameter
    [RENTAL_DESTINATION]

```

- c.** Check the message queue:

```
SQL> select msg from messagequeue order by attime;
```

```
MSG
```

```

-----
Rule 2 matched following primitive events
->Bank Transaction by subject (123302122) of type [TRANSFER]
->Field report refer to(123302122 and 123302123)
=>Recommended Action : Action Type [ADD2WATCHLIST] Action Parameter
[NYPD]

Rule 1 matched following primitive events
->Bank Transaction by subject (123302122) of type [TRANSFER]
->Transportation by subject(123302122) use vessel [TRUCK]
=>Recommended Action : Action Type [ADD2WATCHLIST] Action Parameter
[NYPD]

Rule 2 matched following primitive events
->Transportation by subject(123302122) use vessel [TRUCK]
->Field report refer to(123302122 and 123302123)
=>Recommended Action : Action Type [ADD2WATCHLIST] Action Parameter
[NYPD]

```

```

Rule 2 matched following primitive events
->Bank Transaction by subject (123302122) of type [TRANSFER]
->Transportation by subject(123302122) use vessel [TRUCK]
=>Recommended Action : Action Type [ADD2WATCHLIST] Action Parameter
[NYPD]

```

```

Rule 3 matched following primitive events
->Bank Transaction by subject (123302122) of type [TRANSFER]
->Transportation by subject(123302122) use vessel [TRUCK]
=>Recommended Action : Action Type [STARTBACKGROUNDCHECK] Action
Parameter [RENTAL_DESTINATION]

```

d. Truncate the table messagequeue:

```
SQL> truncate table messagequeue;
```

e. Now lets assume you sleep past the deadline for rule 4. The scheduler process picks up this rule and executes its action. The result is a new message in the message queue.

```
SQL> exec dbms_lock.sleep(180);
```

f. The following action is executed for rule 4 after the deadline time is elapsed:

```
SQL> select mesg from messagequeue;
```

```
MESG
```

```

-----
Rule 4 matched following primitive events
->Bank Transaction by subject (123302122) of type [TRANSFER]
->Transportation by subject(123302122) use vessel [TRUCK]
=>Recommended Action : Action Type [ADD2HIGH_RISK_AREA] Action
Parameter [RENTAL_DESTINATION]

```

9.2 Order Management Rules Application

This Order Management rules application demonstrates the use of Rules Manager for the event data that is stored in relational tables.

The basic steps to create the Order Management rules application with composite events are as follows:

1. Create the three relational tables to store the information about the purchase orders, shipping information, and payment information, as follows:

```

create table PurchaseOrders
  (orderId      NUMBER,
   custId      NUMBER,
   itemId      NUMBER,
   itemType    VARCHAR2(30),
   quantity    NUMBER,
   shipBy      DATE);

create table ShipmentInfo
  (orderId      NUMBER,
   destState   VARCHAR2(2),
   address     VARCHAR2(50),
   shipTime    DATE,
   shipType    VARCHAR2(10));

```

```

create table PaymentInfo
  (orderId    NUMBER,
   payType    VARCHAR2(10), -- Credit Card / Check --
   amountPaid NUMBER,
   pymtTime   DATE,
   billState  VARCHAR2(2));

```

2. Create the event structure. The event structures that refer to the existing tables using table alias constructs cannot be created from object types. Instead, model such event structures as Expression Filter attribute sets, as follows:

```

begin
  DBMS_RLMGR.CREATE_EVENT_STRUCTURE (event_struct => 'OrderMgmt');

  DBMS_RLMGR.ADD_ELEMENTARY_ATTRIBUTE(
    event_struct => 'OrderMgmt',
    attr_name    => 'po',
    tab_alias    => RLM$TABLE_ALIASES('PurchaseOrders'));

  DBMS_RLMGR.ADD_ELEMENTARY_ATTRIBUTE(
    event_struct => 'OrderMgmt',
    attr_name    => 'si',
    tab_alias    => RLM$TABLE_ALIASES('ShipmentInfo'));

  DBMS_RLMGR.ADD_ELEMENTARY_ATTRIBUTE(
    event_struct => 'OrderMgmt',
    attr_name    => 'py',
    tab_alias    => RLM$TABLE_ALIASES('PaymentInfo'));
end;
/

```

3. Create the rule class (database table for rules) for the OrderMgmt composite event. Also specify the DMLEVENTS property to process the rules for each inserted row into the event data tables, as follows:

```

BEGIN
  DBMS_RLMGR.CREATE_RULE_CLASS (
    rule_class => 'OrderMgmtRC',
    event_struct => 'OrderMgmt',
    action_cbk => 'OrderMgmtCBK',
    actprf_spec => 'actionType VARCHAR2(40), actionParam VARCHAR2(100)',
    rslt_viewnm => 'MatchingOrders',
    rlcls_prop => '<composite
                  equal="po.orderId, si.orderId, py.orderId"
                  dmlevents="I"/>');
END;
/

```

This step also creates the skeleton for an action callback procedure with the specified name, as follows:

```

desc OrderMgmtCBK;
PROCEDURE OrderMgmtCBK

```

Argument Name	Type	In/Out	Default?
PO	ROWID	IN	
SI	ROWID	IN	
PY	ROWID	IN	
RLM\$RULE	RECORD	IN	
RLM\$RULEID	VARCHAR2(100)	IN	

ACTIONTYPE	VARCHAR2(40)	IN
ACTIONPARAM	VARCHAR2(100)	IN
RLM\$RULECOND	VARCHAR2(4000)	IN
RLM\$RULEDESC	VARCHAR2(1000)	IN

4. Implement the callback procedure to perform the appropriate action for each matching rule, based on the event instances that matched the rule and the action preferences associated with the rule. In this case, a message displayed to the screen is considered one action, as shown in the following example:

```
CREATE OR REPLACE PROCEDURE OrderMgmtCBK (
  po      ROWID, -- rowid from the PurchaseOrders table
  si      ROWID, -- rowid from the ShipmentInfo table
  py      ROWID, -- rowid from the PaymentInfo table
  rlm$rule OrderMgmtRC%ROWTYPE) IS
  ordId   NUMBER;
  msg     VARCHAR2(2000);
begin
  -- the rowid arguments represent the primitive events that are
  -- rows inserted into the corresponding tables. Use the rowids
  -- to fetch necessary values.
  if (po is not null) then
    select orderId into ordId from PurchaseOrders where rowid = po;
  elsif (si is not null) then
    select orderId into ordId from ShipmentInfo where rowid = si;
  elsif (py is not null) then
    select orderId into ordId from PaymentInfo where rowid = py;
  end if;

  msg := 'Order number: ' || ordId || ' Matched rule: '
        || rlm$rule.rlm$ruleid || chr(10) ||
        '-> Recommended Action : ' || chr(10) ||
        '      Action Type [' || rlm$rule.actionType ||
        ']' || chr(10) || '      Action Parameter [' ||
        rlm$rule.actionParam || ']';

  dbms_output.put_line (msg || chr(10));
end;
/
```

5. Add user-defined functions that may be useful in rule conditions:

```
create or replace function getCustType(custId number)
  return VARCHAR2 is
begin
  -- the actual function implementation can rely on other
  -- relational tables to derive the customer type information
  return 'GOLD';
end;
/

exec DBMS_RLMGR.ADD_FUNCTIONS('OrderMgmt', 'getCustType');
```

6. Add some rules:

- a. Rule: If the order is for more than 100 routers and the payment is received as a check, contact the customer to update the status of the order. Note that the join predicate across event types is specified at the rule class level. For example:

```
INSERT INTO OrderMgmtRC (rlm$ruleid, actionType, actionParam, rlm$rulecond)
VALUES (1, 'CALL_CUSTOMER', 'UPDATE_ORDER_STATUS',
```

```
'<condition>
  <and>
    <object name="po">
      itemType = 'ROUTER' and quantity > 100
    </object>
    <object name="py">
      payType = 'CHECK'
    </object>
  </and>
</condition>');
```

- b.** Rule: If the order is placed by a Gold customer, and the items are shipped before receiving a payment, adjust the customer's credit. For example:

```
INSERT INTO OrderMgmtRC (rlm$ruleid, actionType, actionParam, rlm$rulecond)
VALUES (2, 'UPDATE_CUST_PROFILE', 'DECR_AVAILABLE_CREDIT',
  '<condition>
    <and>
      <object name="po"> getCustType(custid) = 'GOLD' </object>
      <object name="si"/>
      <not>
        <object name="py"/>
      </not>
    </and>
  </condition>');
```

- c.** Rule: If the order is placed by a Gold customer and the item is shipped within 1 day prior to the shipby date, increment the quality of service statistics. For example:

```
INSERT INTO OrderMgmtRC (rlm$ruleid, actionType, actionParam, rlm$rulecond)
VALUES (3, 'UPDATE_STATISTICS', 'INCREMENT_QOS',
  '<condition>
    <and join="po.shipby > si.shiptime-1">
      <object name="po"> getCustType(custid) = 'GOLD' </object>
      <object name="si"/>
    </and>
  </condition>');
```

- 7.** Process the rules for some primitive events, which are the rows inserted into corresponding data tables, as shown in the following list:

- a.** The following event partially matches some of the rules in the rule class and does not result in any action:

```
insert into PurchaseOrders (orderId, custId, itemId, itemType,
                           quantity, shipBy) values
(1, 123, 234, 'ROUTER', 120, '01-OCT-2004');
```

- b.** The following event in combination with the previous added events matches two rules and fires the corresponding actions:

```
insert into ShipmentInfo (orderId, deststate, address, shipTime,
                          shipType) values
(1, 'CA', '1 Main street, San Jose', '29-SEP-2004', '1 Day Air');
Order number: 1 Matched rule: 2
-> Recommended Action :
    Action Type [UPDATE_CUST_PROFILE]
    Action Parameter [DECR_AVAILABLE_CREDIT]
```

```
Order number: 1 Matched rule: 3
-> Recommended Action :
```

```

Action Type [UPDATE_STATISTICS]
Action Parameter [INCREMENT QOS]

```

c. The following event matches one more rule:

```

insert into PaymentInfo (orderId, paytype, amountpaid, pymttime,
                        billstate) values
(1, 'CHECK', 100000, '30-SEP-2004', 'CA');
Order number: 1 Matched rule: 1
-> Recommended Action :
    Action Type [CALL_CUSTOMER]
    Action Parameter [UPDATE_ORDER_STATUS]

```

Now, consider a similar application without the use of the DMLEVENTS property. This implies that the user explicitly invokes the Rules Manager APIs to process the rules for some data stored in relational tables. This rules class shares the event structure with the OrderMgmtRC rule class.

1. Create the rule class (database table for rules) for the OrderMgmtRC2 composite event, as follows:

```

BEGIN
  DBMS_RLMGR.CREATE_RULE_CLASS (
    rule_class    => 'OrderMgmtRC2',
    event_struct  => 'OrderMgmt',
    action_cbk    => 'OrderMgmtCBK2',
    actprf_spec   => 'actionType VARCHAR2(40), actionParam VARCHAR2(100)',
    rslt_viewnm   => 'MatchingOrders2',
    rlcls_prop    => '<composite equal="po.orderId, si.orderId, py.orderId"/>');
END;
/

```

2. Implement the callback procedure to perform the appropriate action for each matching rule, based on the event instances that matched the rule and the action preferences associated with the rule, as follows:

```

--- Implement the action callback procedure --
CREATE OR REPLACE PROCEDURE OrderMgmtCBK2 (
  po      ROWID, -- rowid from the PurchaseOrders table
  si      ROWID, -- rowid from the ShipmentInfo table
  py      ROWID, -- rowid from the PaymentInfo table
  rlm$rule OrderMgmtRC2%ROWTYPE) IS
  ordId   NUMBER;
  msg     VARCHAR2(2000);
begin
  -- the rowid argument represent the primitive events that are
  -- rows inseted into the corresponding tables. Use the rowids
  -- to fetch necessary values.
  if (po is not null) then
    select orderId into ordId from PurchaseOrders where rowid = po;
  elsif (si is not null) then
    select orderId into ordId from ShipmentInfo where rowid = si;
  elsif (py is not null) then
    select orderId into ordId from PaymentInfo where rowid = py;
  end if;

  msg := 'Order number: '||ordId||' Matched rule: '
        ||rlm$rule.rlm$ruleid||chr(10)||
        '-> Recommended Action : '||chr(10)||
        '    Action Type ['||rlm$rule.actionType||
        ']'||chr(10)||'    Action Parameter ['||

```

```

        rlm$rule.actionParam||']';

        dbms_output.put_line (msg||chr(10));
end;
/

```

3. Insert the same set of rules into the new rule class, as follows:

```

insert into OrderMgmtRC2 (select * from OrderMgmtRC);
commit;

```

4. Process the rules for the rows in the data tables as shown in the code that follows. Because DML events are not configured for this rule class, the application must explicitly process the rules for the rows in the data table. The ROWIDS of the rows inserted into the data tables are used as references to the events and they are passed to the PROCESS_RULES procedure to process the rules.

```

var datarid varchar2(40);

insert into PurchaseOrders (orderId, custId, itemId, itemType,
                           quantity, shipBy) values
(2, 123, 234, 'ROUTER', 120, '01-OCT-2004')
returning rowid into :datarid;

BEGIN
  dbms_rlmgr.process_rules (rule_class => 'OrderMgmtRC2',
                           event_type => 'PurchaseOrders',
                           event_inst => :datarid);
END;
/

insert into ShipmentInfo (orderId, deststate, address, shipTime,
                          shipType) values
(2, 'CA', '1 Main street, San Jose', '29-SEP-2004', '1 Day Air')
returning rowid into :datarid;

BEGIN
  dbms_rlmgr.process_rules (rule_class => 'OrderMgmtRC2',
                           event_type => 'ShipmentInfo',
                           event_inst => :datarid);
END;
/
Order number: 2 Matched rule: 2
-> Recommended Action :
      Action Type [UPDATE_CUST_PROFILE]
      Action Parameter [DECR_AVAILABLE_CREDIT]

Order number: 2 Matched rule: 3
-> Recommended Action :
      Action Type [UPDATE_STATISTICS]
      Action Parameter [INCREMENT_QOS]

insert into PaymentInfo (orderId, paytype, amountpaid, pymttime,
                         billstate) values
(2, 'CHECK', 100000, '30-SEP-2004', 'CA')
returning rowid into :datarid;

BEGIN
  dbms_rlmgr.process_rules (rule_class => 'OrderMgmtRC2',
                           event_type => 'PaymentInfo',
                           event_inst => :datarid);

```

```

END;
/
Order number: 2 Matched rule: 1
-> Recommended Action :
    Action Type [CALL_CUSTOMER]
    Action Parameter [UPDATE_ORDER_STATUS]

```

Now, try the session oriented evaluation of rules where the results from matching rules are available in the results view to be queried, as shown in the following list:

1. A description of the result view table follows:

```

set linesize 80;
desc MatchingOrders2;

```

Name	Null?	Type
RLM\$EVENTID		ROWID
PO		ROWID
SI		ROWID
PY		ROWID
RLM\$RULEID		VARCHAR2 (100)
ACTIONTYPE		VARCHAR2 (40)
ACTIONPARAM		VARCHAR2 (100)
RLM\$RULECOND		VARCHAR2 (4000)
RLM\$RULEDESC		VARCHAR2 (1000)

```

select count(*) from MatchingOrders2;

COUNT(*)
-----
0

```

2. Process the rules for the rows in the data tables. Note the use of the `ADD_EVENT` procedure instead of the `PROCESS_RULES` procedure in the previous example. This ensures that the results from the matching events with the rules are stored in the rule class results view. For example:

```

insert into PurchaseOrders (orderId, custId, itemId, itemType,
                           quantity, shipBy) values
(3, 123, 234, 'ROUTER', 120, '01-OCT-2004')
returning rowid into :datarid;

--- Use ADD_EVENT API in the place of PROCESS_RULES ---
BEGIN
  dbms_rlmgr.add_event (rule_class => 'OrderMgmtRC2',
                       event_type => 'PurchaseOrders',
                       event_inst => :datarid);
END;
/

insert into ShipmentInfo (orderId, deststate, address, shipTime,
                         shipType) values
(3, 'CA', '1 Main street, San Jose', '29-SEP-2004', '1 Day Air')
returning rowid into :datarid;

BEGIN
  dbms_rlmgr.add_event (rule_class => 'OrderMgmtRC2',
                       event_type => 'ShipmentInfo',
                       event_inst => :datarid);
END;
/

```

```

insert into PaymentInfo (orderId, paytype, amountpaid, pymttime,
                        billstate) values
(3, 'CHECK', 100000, '30-SEP-2004', 'CA')
returning rowid into :datarid;

BEGIN
  dbms_rlmgr.add_event (rule_class => 'OrderMgmtRC2',
                       event_type => 'PaymentInfo',
                       event_inst => :datarid);
END;
/

```

3. Because the event structure is configured with table aliases, the events are represented using the ROWIDS from the corresponding tables, as follows:

```

column rlm$ruleid format a7;
column actiontype format a25;
column actionparam format a25;
select po, si, py, rlm$ruleid, actionType, actionParam from MatchingOrders2;

```

PO	SI	PY	RLM\$RUL

ACTIONTYPE	ACTIONPARAM		

AAAOBxAAEAAAAHPAAC	AAAOByAAEAAAAHXAAC		2
UPDATE_CUST_PROFILE	DECR_AVAILABLE_CREDIT		
AAAOBxAAEAAAAHPAAC	AAAOByAAEAAAAHXAAC		3
UPDATE_STATISTICS	INCREMENT QOS		
AAAOBxAAEAAAAHPAAC	AAAOBzAAEAAAAHFAAC		1
CALL_CUSTOMER	UPDATE_ORDER_STATUS		

4. The ROWIDS can be used to derive the actual event values from the data tables, as follows:

```

select
  (select orderId from purchaseOrders where rowid = po) as OrderId,
  rlm$ruleid, actionType, actionParam from MatchingOrders2;

```

ORDERID	RLM\$RUL	ACTIONTYPE	ACTIONPARAM

3	2	UPDATE_CUST_PROFILE	DECR_AVAILABLE_CREDIT
3	3	UPDATE_STATISTICS	INCREMENT QOS
3	1	CALL_CUSTOMER	UPDATE_ORDER_STATUS

Part II

Expression Filter

This part introduces developing applications using Expression Filter feature.

Part II contains the following chapters:

- [Chapter 10, "Oracle Expression Filter Concepts"](#)
- [Chapter 11, "Indexing Expressions"](#)
- [Chapter 12, "Expressions with XPath Predicates"](#)
- [Chapter 13, "Expressions with Spatial Predicates"](#)
- [Chapter 14, "Using Expression Filter with Utilities"](#)
- [Chapter 15, "SQL Operators and Statements"](#)
- [Chapter 16, "Object Types"](#)
- [Chapter 17, "Management Procedures Using the DBMS_EXPFIL Package"](#)
- [Chapter 18, "Expression Filter Views"](#)

Oracle Expression Filter Concepts

Oracle Expression Filter, a feature of Oracle Database 10g, is a component of Rules Manager 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 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.

10.1 What Is Expression Filter?

Expression Filter provides a data type, operator, and index type to store, evaluate, and index expressions that describe an interest in a data item or piece of information. See *Oracle Database Data Cartridge Developer's Guide* for an explanation of these terms. 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:

- Attribute set: a definition of the event and its set of attributes
- Expression data type: A virtual data type 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: An index that enhances performance of the EVALUATE operator for large expression sets. Expression indexing is available in Oracle Database Enterprise Edition

10.1.1 Expression Filter Usage Scenarios

This section provides 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 data type. 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 data type) 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 exist 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 the SQL `WHERE` clause) describes an interest in incoming data items.
- The expressions compare attributes to values using relational operators (`=`, `!=`, `<`, `>`, and so on).

10.2 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, functions supplied by Oracle, 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` function supplied by Oracle 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 data type. The values stored in a column of this type are constrained to be expressions. (See [Section 10.2.2](#).) 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 data types 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.

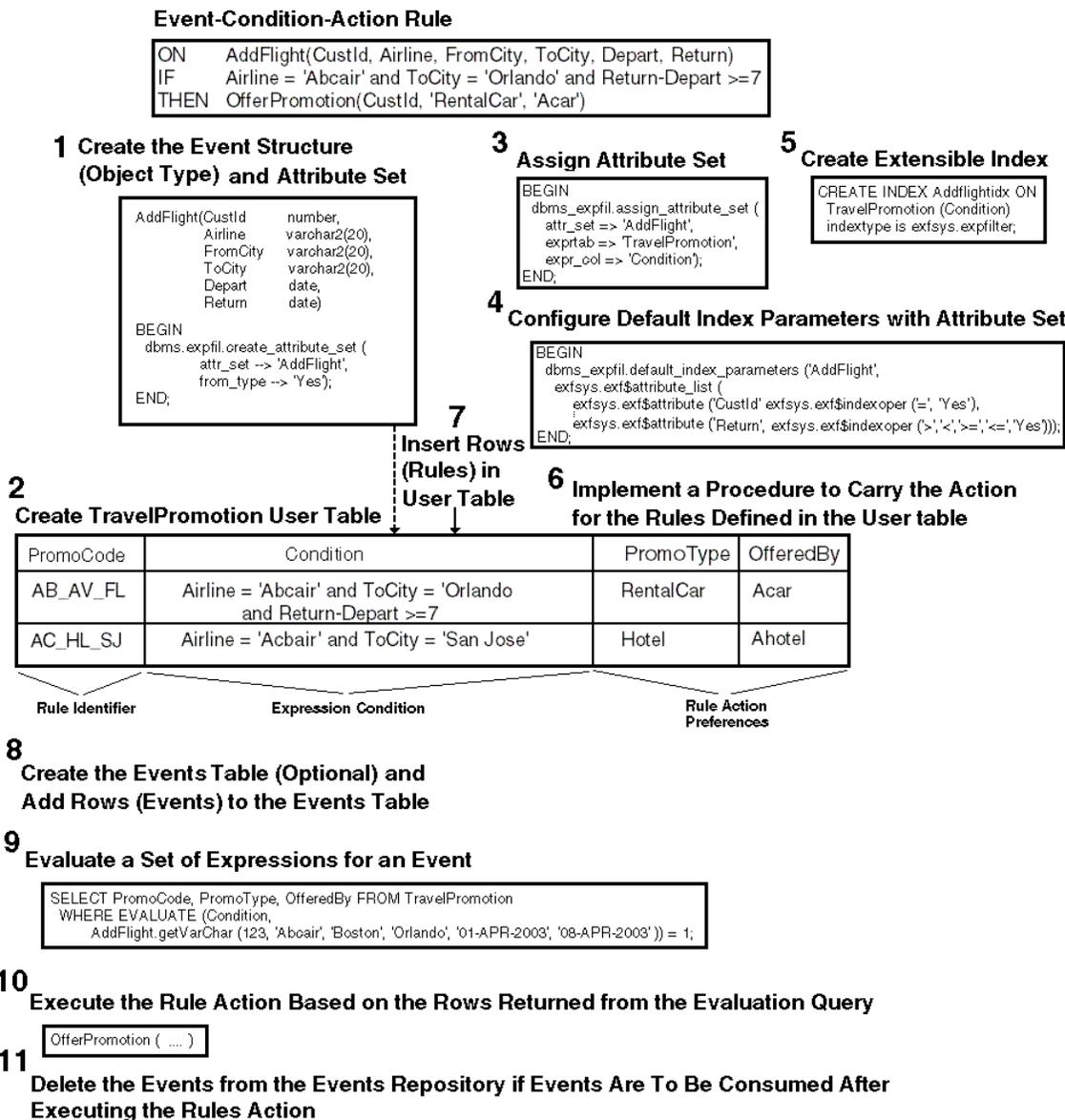
There are four basic steps to create and use an Expression column:

1. Define an attribute set. See [Section 10.2.1](#).
2. Define an Expression column in a user table. See [Section 10.2.2](#).

3. Insert expressions in the table. See [Section 10.2.3](#).
4. Apply the SQL EVALUATE operator to compare expressions to incoming data items. See [Section 10.3](#).

Figure 10–1 shows the process steps for creating and implementing a rules application based on Expression Filter. The remaining sections in this chapter guide you through this procedure.

Figure 10–1 Expression Filter Implementation Process for a Rules Application



10.2.1 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 SQL functions supplied by Oracle 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 10-1](#).
- 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 10-2](#).

If the expressions refer to user-defined functions, you must add the functions to the corresponding attribute set, using the `ADD_FUNCTIONS` procedure of the `DBMS_EXPFIL` package. See [Example 10-3](#).

Attribute Set Examples

[Example 10-1](#) shows how to use an existing object type to create an attribute set. It uses the `CREATE_ATTRIBUTE_SET` procedure.

Example 10-1 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`.

[Example 10-2](#) 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 10-2 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',
                                attr_defv1 => '2000');
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_ATTRIBUTES` Procedure.

If the expressions refer to user-defined functions, you must add the functions to the corresponding attribute set. [Example 10-3](#) shows how to add user-defined functions, using the `ADD_FUNCTIONS` procedure, to an attribute set.

Example 10-3 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 using 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.

To drop an attribute set, you use the `DROP_ATTRIBUTE_SET` procedure. For more information, see `DROP_ATTRIBUTE_SET` Procedure.

10.2.2 Defining Expression Columns

Expression is a virtual data type. 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 data type 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, named `Interest`, that will be used with an attribute set:

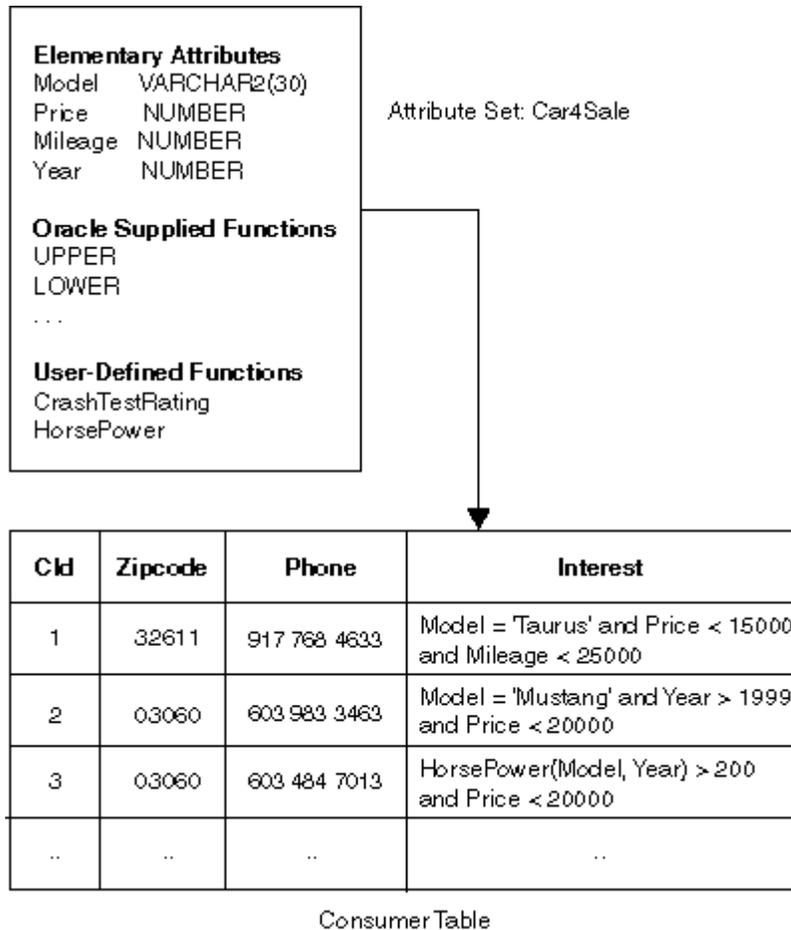
```
CREATE TABLE Consumer (CId          NUMBER,
                       Zipcode      NUMBER,
                       Phone         VARCHAR2(12),
                       Interest      VARCHAR2(200));
```

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`.

[Figure 10-2](#) is a conceptual image of consumers' interests (in trading cars) being captured in a `Consumer` table.

Figure 10–2 Expression Data Type

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`.

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`.

To copy an attribute set across schemas, you use the `COPY_ATTRIBUTE_SET` procedure of the `DBMS_EXPFIL` package. See `COPY_ATTRIBUTE_SET Procedure`.

10.2.3 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 10.4](#).

[Example 10–4](#) shows how to insert an expression (the consumer's interest in trading cars, which is depicted in [Figure 10–2](#)) into the `Consumer` table using the SQL `INSERT` statement.

Example 10–4 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 10–3](#)). [Example 10–5](#) shows how to insert an expression with a reference to a user-defined function, `HorsePower`, into the `Consumer` table.

Example 10–5 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 bulk loaded into an Expression column using SQL*Loader. For more information about bulk loading, see [Section 14.1](#).

10.3 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` data types 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 object type supplied by Oracle that can hold instances of any Oracle data type, both supplied by Oracle 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.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;
```

Note: 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 15](#). For additional examples of the SQL `EVALUATE` operator, see [Appendix B](#).

10.4 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.

10.5 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`.

10.6 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.

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, `EXFSYS.EXPFILTER`, 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 11.3](#)), index processing ([Section 11.4](#)), and user commands for creating and tuning indexes ([Section 11.6](#)).

11.1 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 among 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)`.

11.2 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. A predicate with a BETWEEN operator is treated as two predicates with binary operators, one with the '>=' operator and another with the '<=' operator. See the section about EXF\$INDEXOPER for more information about the BETWEEN operator.

11.3 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. However, 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 11-1](#).

Figure 11–1 Conceptual Predicate Table

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

Rid	G1		G2		G3		Sparse_predicate
	Op	RHS	Op	RHS	Op	RHS	
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 11–1](#), 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 11.4](#).

11.4 Index Processing

To evaluate a data item for a set of expressions, the left-hand side of each predicate group in the data item 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 11–1](#)), 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. See `EXF$INDEXOPER` for more information. 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.

11.5 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.

11.6 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:

1. One-time computation of the left-hand side of the predicate group
2. 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:

1. One-time computation of the left-hand side of the predicate group
2. 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:

1. Parse the subexpression representing the sparse predicates for all the expressions remaining in the working set.
2. 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 and ["CREATE INDEX"](#) in [Chapter 15](#).

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 and "CREATE INDEX" in [Chapter 15](#).

See [Chapter 12](#) 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 and "CREATE INDEX" in [Chapter 15](#).

11.7 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. Beginning with release 10gR2, the selectivity and the cost of an Expression Filter index are computed when statistics are collected on the expression column, the index, or the table storing expressions. These statistics are stored in the Expression Filter dictionary and are used to determine the optimal execution plan for the query with an `EVALUATE` operator.

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

11.8 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 15](#).) 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 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 15](#).

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 10](#). 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.

12.1 Using XPath Predicates in Expressions

Using the `XMLType` data type supplied by Oracle, users can apply XPath predicates on XML documents within a standard SQL `WHERE` clause of a query. These predicates use SQL operators such as `EXTRACT` and `EXISTS` on an instance of the `XMLType` data type 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` data type, 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.

12.2 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.

12.2.1 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 the last phase of expression filtering. For more information about sparse predicates, see [Section 11.4](#).

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 `EXISTS` 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 in the following XPath predicate:

```
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.

12.2.2 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 a value filter, these columns capture the constants appearing with the tag in the node tests and their relational operators.

Note: Only equality operators are indexed in this release.

Figure 12–1 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 11–1. 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 12–1 Conceptual Predicate Table with XPath Predicates

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

12.2.3 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 (see Figure 12–1), 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 are 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 11.4](#)). 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.

12.2.4 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 `DBMS_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.

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.

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>
```

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

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

Expressions with Spatial Predicates

Note: The Oracle Spatial or the Locator components must be installed to use spatial predicates in stored expressions.

The expressions stored in a column of a table may contain spatial predicates defined on `SDO_GEOMETRY` attributes. This section describes an application for spatial predicates using the Car4Sale example introduced in [Chapter 10](#). For this purpose, the information published for each car going on sale includes a `Location` attribute in addition to the `Model`, `Price`, `Mileage`, and `Year` attributes. The `Location` attribute contains geographical coordinates for the vehicle's location, as an instance of the `SDO_GEOMETRY` data type.

Using the `Location` attribute, the consumer interested in a vehicle can restrict the search only to the vehicles that are within a specified distance, say half a mile, of his own location. This can be specified using the following spatial predicate involving the `SDO_WITHIN_DISTANCE` operator:

```
SDO_WITHIN_DISTANCE (
  Location,
  SDO_GEOMETRY (
    2001, 8307,
    SDO_POINT_TYPE(-77.03644, 37.89868, NULL), NULL, NULL
  ) ,
  'distance=0.5 units=mile'
) = 'TRUE'
```

Note that spatial predicates are efficiently evaluated with the help of spatial indexes. [Section 13.1](#) and [Section 13.2](#) will describe how to specify spatial predicates in arbitrary expressions and how to ensure the predicates are evaluated using appropriate spatial indexes.

13.1 Using Spatial Predicates in Expressions

Using the Oracle supplied `SDO_GEOMETRY` data type, users can specify spatial predicates on instances of spatial geometries within a standard SQL `WHERE` clause of a query. These predicates use operators such as `SDO_WITHIN_DISTANCE` and `SDO_RELATE` on an instance of `SDO_GEOMETRY` data type to relate two spatial geometries in a specific way. For more information, see *Oracle Spatial User's Guide and Reference*.

To allow spatial predicates in an expression set, the corresponding attribute set should be created with an attribute of `MDSYS.SDO_GEOMETRY` data type as shown in the following example:

```
CREATE OR REPLACE TYPE Car4Sale AS OBJECT
```

```

        (Model    VARCHAR2(20),
         Year     NUMBER,
         Price    NUMBER,
         Mileage  NUMBER,
         Location MDSYS.SDO_GEOMETRY);
/
BEGIN
  dbms_expfil.create_attribute_set (attr_set => 'Car4Sale',
                                   from_type => 'YES');
END;
/

```

In order to specify predicates on the spatial attribute and index them for efficiency, the geometry metadata describing the dimension, lower and upper bounds, and tolerance in each dimension should be associated with each spatial geometry attribute in the attribute set. This metadata information can be inserted into the USER_SDO_GEOM_METADATA view using the attribute set name in the place of the table name. For more information on the USER_SDO_GEOM_METADATA view and its semantics, see *Oracle Spatial User's Guide and Reference*.

```

INSERT INTO user_sdo_geom_metadata VALUES ('CAR4SALE', 'LOCATION',
      mdsys.sdo_dim_array(
        mdsys.sdo_dim_element('X', -180, 180, 0.5),
        mdsys.sdo_dim_element('Y', -90, 90, 0.5)), 8307);

```

The expression set using the attribute set with one or more SDO_GEOMETRY attributes can include predicates on such attributes using SDO_WITHIN_DISTANCE or SDO_RELATE operators, as shown in the following examples:

```

Model = 'Taurus' and Price < 15000 and Mileage < 25000 and
  SDO_WITHIN_DISTANCE (Location,
    SDO_GEOMETRY(2001, 8307,
      SDO_POINT_TYPE(-77.03644, 37.89868, NULL), NULL, NULL),
    'distance=0.5 units=mile') = 'TRUE'

```

```

Model = 'Taurus' and Price < 15000 and Mileage < 25000 and
  SDO_RELATE (Location,
    SDO_GEOMETRY(2001, 8307, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 3),
      SDO_ORDINATE_ARRAY(-77.03644, 37.89868, -75, 39),
    'mask=anyinteract') = 'TRUE'

```

Note that unlike in the case of expressions with purely non-spatial predicates, expressions with spatial predicates cannot be evaluated when an Expression Filter index is not defined for the expression set. Once an Expression Filter index is created on the column storing the expressions, expressions with spatial predicates can be processed for a data item by passing an instance of SDO_GEOMETRY data type for the Location attribute, along with other attribute values, to the EVALUATE operator.

```

SELECT * FROM Consumer WHERE
  EVALUATE (Interest,
    sys.anyData.convertObject(
      Car4Sale('Mustang', 2002, 20000, 250000,
        SDO_GEOMETRY(2001, 8307,
          sdo_point_type(-77.03644, 38.9059284, null), null, null)))
    ) = 1;

```

The previous query identifies all the rows with expressions that are true based on their spatial and not-spatial predicates.

13.2 Indexing Spatial Predicates

The spatial predicates in the stored expressions are processed using some custom spatial indexes created on the geometries specified in the spatial predicates. These spatial indexes are automatically created when the Expression Filter index is created on the column storing expressions. The expressions with spatial predicates cannot be processed in the absence of these spatial indexes and hence an Expression Filter index is always required to evaluate such expressions.

When an Expression Filter index is defined on an expression column, the spatial attributes in corresponding attribute set are all considered as indexed predicate groups. The predicate table has columns of `SDO_GEOMETRY` type for each of these attributes and spatial indexes are created on these columns. The values stored in an `SDO_GEOMETRY` column of the predicate table are computed based on the values specified in the spatial predicates involving corresponding attribute.

When the expressions are evaluated for a data item, the spatial indexes created on the geometry column in combination with bitmap indexes created for the other indexed predicate groups filter out the expressions that are false based on all indexed predicate groups. The expressions remaining in the working set are further evaluated based on the stored predicate groups and sparse predicates to identify all the expressions that are true for the given data item.

Using Expression Filter with Utilities

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

14.1 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 data type.

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.

14.2 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.

14.2.1 Exporting and Importing Tables Containing Expression Columns

The following guidelines and known behavior associated with exporting and importing tables containing Expression columns will assist you in this operation.

- 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 data type 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 `SQL ALTER INDEX ... REBUILD` statement.

14.2.2 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.

14.2.3 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.

SQL Operators and Statements

This chapter provides reference information about the SQL `EVALUATE` operator and SQL statements used to index expression data. [Table 15–1](#) lists the statements and their descriptions. For complete information about SQL statements, see *Oracle Database SQL Reference*.

Table 15–1 *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 a 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` data types 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 [Section 10.3](#) 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 [Section 11.8](#) 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

Lists the stored attributes for the Expression Filter index.

INDEXATTRS

Lists 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.) 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 XPIINDEX_PARAMETERS Procedure sections in [Chapter 17](#) and the "USER_EXPFIL_INDEX_PARAMS View" in [Chapter 18](#).)

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 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 11](#) 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;
```

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. [Table 16-1](#) describes the Expression Filter object types.

None of the values and names passed to the types defined in this chapter are case sensitive. To preserve the case, use double quotation marks around the values.

Tip: See the "Expression Filter Types" chapter in *Oracle Database PL/SQL Packages and Types Reference* for all reference information concerning Expression Filter object types.

Table 16-1 Expression Filter Object Types

Object Type Name	Description
<code>EXF\$ATTRIBUTE</code>	Specifies the stored and indexed attributes for the Expression Filter indexes
<code>EXF\$ATTRIBUTE_LIST</code>	Specifies a list of stored and indexed attributes when configuring index parameters
<code>EXF\$INDEXOPER</code>	Specifies a list of common operators in predicates with a stored or an indexed attribute
<code>EXF\$TABLE_ALIAS</code>	Indicates a special form of elementary attribute used to manage expressions defined on one or more database tables
<code>EXF\$XPath_TAG</code>	Configures an XML element or an XML attribute for indexing a set of XPath predicates
<code>EXF\$XPath_TAGS</code>	Specifies a list of XML tags when configuring the Expression Filter index parameters



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 17-1 describes the procedures in the DBMS_EXPFIL package. These procedures are further described in this chapter.

None of the values and names passed to the procedures defined in the DBMS_EXPFIL package are case sensitive, unless otherwise mentioned. To preserve the case, use double quotation marks around the values.

Tip: See the "DBMS_EXPFIL" chapter in *Oracle Database PL/SQL Packages and Types Reference* for all reference information concerning Expression Filter package procedures.

Table 17-1 DBMS_EXPFIL Procedures

Procedure	Description
ADD_ELEMENTARY_ATTRIBUTE Procedure	Adds the specified attribute to the attribute set
ADD_FUNCTIONS Procedure	Adds a function, type, or package to the approved list of functions with an attribute set
ASSIGN_ATTRIBUTE_SET Procedure	Assigns an attribute set to a column storing expressions
BUILD_EXCEPTIONS_TABLE Procedure	Creates an exception table to hold references to invalid expressions
CLEAR_EXPRSET_STATS Procedure	Clears the predicate statistics for an expression set
COPY_ATTRIBUTE_SET Procedure	Makes a copy of the attribute set
CREATE_ATTRIBUTE_SET Procedure	Creates an attribute set
DEFAULT_INDEX_PARAMETERS Procedure	Assigns default index parameters to an attribute set
DEFAULT_XPINDEX_PARAMETERS Procedure	Assigns default XPath index parameters to an attribute set
DEFRAG_INDEX Procedure	Rebuilds the bitmap indexes online to reduce fragmentation
DROP_ATTRIBUTE_SET Procedure	Drops an unused attribute set
GET_EXPRSET_STATS Procedure	Collects predicate statistics for an expression set
GRANT_PRIVILEGES Procedure	Grants an expression DML privilege to a user
INDEX_PARAMETERS Procedure	Assigns index parameters to an expression set

Table 17-1 (Cont.) DBMS_EXPFIL Procedures

Procedure	Description
MODIFY_OPERATOR_LIST	Modifies the list of common operators associated with a certain attribute in the attribute set
REVOKE_PRIVILEGE Procedure	Revokes an expression DML privilege from a user
UNASSIGN_ATTRIBUTE_SET Procedure	Breaks the association between a column storing expressions and the attribute set
VALIDATE_EXPRESSIONS Procedure	Validates expression metadata and the expressions stored in a column
XPINDEX_PARAMETERS Procedure	Assigns XPath index parameters to an expression set

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 18–1 lists the names of the views and their descriptions.

Table 18–1 Expression Filter Views

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

18.1 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	Data Type	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 the schema extension)
OBJECT_OWNER	VARCHAR2	Owner of the function or package (or type)
OBJECT_NAME	VARCHAR2	Name of the function or package (or type)

Column Name	Data Type	Description
OBJECT_TYPE	VARCHAR2	Type of the object at the time the object was added to the attribute set: <ul style="list-style-type: none"> ▪ 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

18.2 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	Data Type	Description
ATTRIBUTE_SET_NAME	VARCHAR2	Name of the attribute set.
ATTRIBUTE	VARCHAR2	Name of the elementary attribute.
DATA_TYPE	VARCHAR2	Data type of the attribute.
ASSOCIATED_TABLE	VARCHAR2	Name of the corresponding table for the table alias attribute. Null for all other types of attributes.

18.3 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	Data Type	Description
ATTRIBUTE_SET_NAME	VARCHAR2	Name of the attribute set

18.4 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	Data Type	Description
ATTRIBUTE_SET_NAME	VARCHAR2	Name of the attribute set
ATTRIBUTE	VARCHAR2	Name of the stored attribute
DATA_TYPE	VARCHAR2	Data type of the attribute
ELEMENTARY	VARCHAR2	YES, if the attribute is also the elementary attribute of the attribute set; otherwise, NO

Column Name	Data Type	Description
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

18.5 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	Data Type	Description
EXPR_TABLE	VARCHAR2	Name of the table storing expressions
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)

18.6 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	Data Type	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 left-hand side (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

Column Name	Data Type	Description
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
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

18.7 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	Data Type	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	Data type 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

18.8 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	Data Type	Description
INDEX_NAME	VARCHAR2	Name of the index
PREDICATE_TABLE	VARCHAR2	Name of the predicate table used for the index

Column Name	Data Type	Description
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: <ul style="list-style-type: none"> ▪ 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

18.9 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	Data Type	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
SUBEXPRESSION	VARCHAR2	The arithmetic expression that represents the stored attribute (also the LHS of predicates in the set)
DATA_TYPE	VARCHAR2	Derived data type 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

18.10 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	Data Type	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

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;
```

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 can 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.

Internal Objects

The Expression Filter and Rules Manager features use 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$` prefix (Expression Filter) or the `RLM$` prefix (Rules Manager) and are maintained using either the Expression Filter or Rules Manager APIs. You should not modify these objects.

C.1 Attribute Set or Event Structure Object Type

The Expression Filter maintains the concept of an attribute set through an object type with a matching name, while Rules Manager maintains the concept of an event structure through an object type with a matching name. The object type used for an attribute set or event structure may not contain any user methods, and it should not be an evolved type (with the use of the SQL `ALTER TYPE` command). If the attribute set or event structure is not created from an existing object type, Expression Filter or Rules Manager creates the object type with the matching name and maintains it throughout the life of the attribute set or event structure. 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 and Rules Manager create 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 or event structure 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 10](#):

```
-- 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;
```

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

C.2 Expression Filter Internal Objects

[Section C.2.1](#), [Section C.2.2](#), and [Section C.2.3](#) describe some Expression Filter specific information about Expression Filter internal objects.

C.2.1 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.

C.2.2 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 11.8](#).

C.2.3 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
- Drop the attribute set and associated metadata when the user is dropped with a `CASCADE` option
- 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.

Converting Rules Applications

This appendix describes differences between Expression Filter and Rules Manager and how to convert an Expression Filter rules application to a Rules Manager rules application.

D.1 Differences Between Expression Filter and Rules Manager

Before converting your Expression Filter application to a Rules Manager application, you should understand the differences between each feature and some of the reasons why you should use Rules Manager. If you are ready to convert your Expression Filter application to a Rules Manager application, see [Section D.2](#).

Expression Filter is best used to model simple rules-based systems. A simple rules-based system consists of a primitive event that may have a small-to-very large class of rules (hundreds to millions of rules).

Rules Manager is best used for modeling a wide range of rules-based systems from the simplest to the most complex. A simple rules-based system is again a primitive event having a small-to-very large class of rules (hundreds to millions of rules), while a very complex rules-based system may involve many sets of composite events (each consisting of two or more primitive events) each with a very large class of rules (millions of rules) that can represent very complex rule conditions and that enforce event management policies that require reusing primitive events and handling duplicate composite events, and so forth.

[Table D-1](#) shows step-by-step differences between implementing and using the Expression Filter and Rules Manager features that uses a primitive (simple) event.

Table D–1 Implementation Differences Between Expression Filter and Rules Manager for Rules Applications That Use a Primitive (Simple) Event

Expression Filter	Rules Manager
1. Create the event structure and its set of attributes or use an existing object type definition.	1. Create the event structure and its set of attributes or use an existing object type definition.
2. Create a table to store the rule conditions and associated information.	2. Create the rule class for the event structure. <ul style="list-style-type: none"> o This implicitly creates the skeleton for a callback procedure to perform the action. o This implicitly creates a rules class table to store the corresponding rule definitions and rule action preferences. o This defines the results view, if specified in the rule class definition, to temporarily store the results of processing rules.
3. Assign the event structure that is captured as an Expression Filter attribute set to the condition column in the table.	Note: Rules Manager implicitly creates the Expression data type column (<code>rlm\$rulecond</code>) in the generated rule class table.
4. Configure the default index parameters with the attribute set.	Note: Rules Manager implicitly creates the default index parameters with the attribute set.
5. Create an Expression Filter index on the Expression column in the user table.	Note: Rules Manager implicitly creates the Expression Filter indexes on the necessary Expression columns in the rule class table.
6. Implement a procedure to carry the action for the rules defined in the user table.	3. Replace the system generated callback procedure with the user implementation to perform the appropriate rule action for each matching rule.
7. Insert rule conditional expressions and accompanying information to the user table.	4. Insert rules into the rule class table.
8. Create the events table to store the past events if the rules in the user table rely on composite events.	Note: Rules Manager implicitly creates an events table to keep track of the past events until they are no longer required.
9. Apply the SQL <code>EVALUATE</code> operator to compare expressions stored in the Expressions column to the rows stored in the event table	5. Process the rules for an event. Note: Rules Manager automatically applies the SQL <code>EVALUATE</code> operator to compare rule conditions stored in the <code>rlm\$rulecond</code> column of the rules class table to an event instance.
10. Execute the action procedure for one or more rows returned by the previous query.	Note: With the <code>PROCESS_RULES</code> call, Rules Manager implicitly executes the action for the matching rules by invoking the preconfigured action callback procedure.
11. Delete the events from the events table if the application calls for the consumption of the events immediately after executing the rule actions.	Note: Rules Manager can be configured to automatically consume the events by using the appropriate event management policies.

From [Table D–1](#), Rules Manager automatically performs a number of operations (through subsumption of Expression Filter functionality) that normally had to be done manually using Expression Filter. Because many Expression Filter features are implicitly used by Rules Manager, Rules Manager is easier to use and is the recommended choice, especially for complex rules applications involving composite events.

If you have already modeled and implemented a rules-based system application that uses Expression Filter and you want to convert your application to a Rules Manager application, see [Section D.2](#) for a description of this process.

D.2 Converting an Expression Filter Application to a Rules Manager Application

Expression Filter is a component of Rules Manager. Rules Manager is the preferred feature to use for developing rules applications in release Oracle Database 10g Release 2 (10.2). Expression Filter applications developed in release Oracle Database 10g Release 1 (10.1) can be converted to Rules Manager applications once you understand the main differences between these two features relative to the tables storing the expressions or rules. These differences from an implementation perspective are the name of and structure of the user table containing the expression column for Expression Filter applications versus the name of and structure of the rules class table containing the rule condition column and action preference columns for a Rules Manager application.

The process of converting an Expression Filter application to a Rules Manager application is to complete Steps 1 through 3 as described in the Rules Manager column in [Table D-1](#). Then, instead of populating the rules class table using a SQL INSERT statement as shown in Step 4, use the following SQL statement syntax to copy the rows from the Expression Filter user table to the Rules Manager rules class table:

```
SQL INSERT INTO <rules-manger-rules-class-table> (field1, field3, field4, field2)
      SELECT field1, field2, field3, field4 from <expression-filter-user-table>;
```

This SQL INSERT statement syntax populates the Rules Manager rules class table with the expression conditions from the condition Expression column in the Expression Filter user table along with the desired action preference columns. For example, the following SQL statements would perform this operation after executing SQL DESCRIBE statements to view the structure of each of these tables to determine which columns you want to copy and in what order to copy them:

--Assume the Expression Filter user table has the following structure:

```
SQL> DESCRIBE user_exprfiltertable;
```

Name	Null?	Type
ID		VARCHAR2(100)
CONDITION		VARCHAR2(200)
POSTALCODE		VARCHAR2(10)
PHONE		VARCHAR2(10)

--Assume the Rules Manager rules class table has the following structure:

```
SQL> DESCRIBE rm_rules_classtable;
```

Name	NULL?	TYPE
RLM\$RULEID		VARCHAR2(100)
PostalCode		VARCHAR2(10)
Phone		VARCHAR2(10)
RLM\$RULECOND		VARCHAR2(4000)
RLM\$RULEDESC		VARCHAR2(1000)

--Insert statement to use that copies rows from the Expression Filter user table --to the Rules Manager rules class table:

```
INSERT INTO rm_rules_classtable (rlm$ruleid, PostalCode, Phone, rlm$rulecond)
      SELECT ID, PostalCode, Phone, Condition FROM user_exprfiltertable;
```

Once the rules class table is populated with the rows of the Expression Filter user table, proceed to complete Steps 5 through 7 as described in the Rules Manager column in [Table D-1](#). Upon completion of these steps, you will have a Rules Manager rules application.

Note that to adapt Rules Manager to one of your existing applications, you can use this same `SQL INSERT INTO` syntax to populate the rules class table with data residing within other tables of your application, but only after Rules Manager initially creates this table for you. This is the best way to populate the rules class table with the desired values for these same columns that are defined as part of the rule class creation process described in Step 2 in the Rules Manager column in [Table D-1](#). Now you might just be beginning to realize that adapting your existing application to use Rules Manager is a straight-forward process and it is not too difficult to quickly produce results once you understand how to develop rules applications using Rules Manager. The conceptual approach of this development process is described in more detail in [Section 1.2](#).

Installing Rules Manager and Expression Filter

Rules Manager and Expression Filter provide a SQL schema and PL/SQL and Java packages that store, retrieve, update, and query collections of expressions (rule conditions) in an Oracle database.

Rules Manager and Expression Filter are installed automatically with Oracle Database 10g Standard Edition and Oracle Database 10g Enterprise Edition. Each 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 Rules Manager is named `catrul.sql` and is found in the `$ORACLE_HOME/rdbms/admin/` directory. The script to install the Expression Filter is named `catexf.sql` and is found in the `$ORACLE_HOME/rdbms/admin/` directory. These scripts should be executed from a SQL*Plus session while connected as SYSDBA. Rules Manager can be uninstalled using `catnorul.sql` script in the same directory. Expression Filter can be uninstalled using the `catnoexf.sql` script in the same directory. Uninstalling Expression Filter implicitly uninstalls Rules Manager.

The Rules Manager and Expression Filter features are 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 Oracle Database, a demonstration script is installed for both the Rules Manager and Expression Filter features. The scripts `ruldemo.sql` (Rules Manager demo) and `exfdemo.sql` (Expression Filter demo) are located in the `$ORACLE_HOME/rdbms/demo/` directory.

XML Schemas

The following XML Schemas for the rule class properties and the rule conditions can be used to build authoring tools for rule management:

Rule Class Properties

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:xdb="http://xmlns.oracle.com/xdb"
            xmlns:rlmp="http://www.oracle.com/rlmgr/rsetprop.xsd"
            elementFormDefault="qualified"
            targetNamespace="http://www.oracle.com/rlmgr/rsetprop.xsd">
  <xsd:element name="simple" type="rlmp:SimpleRuleSetProp"/>
  <xsd:element name="composite" type="rlmp:CompositeRuleSetProp">
    <xsd:unique name="objtype">
      <xsd:selector xpath="./object"/>
      <xsd:field xpath="@type"/>
    </xsd:unique>
  </xsd:element>

  <!-- Properties of a rule class with simple events -->
  <xsd:complexType name="SimpleRuleSetProp">
    <xsd:complexContent>
      <xsd:restriction base="xsd:anyType"> <!-- empty element -->
        <xsd:attribute name="ordering" type="xsd:string"/>
        <xsd:attribute name="storage" type="xsd:string"/>
        <xsd:attribute name="autocommit">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="yes"/>
              <xsd:enumeration value="no"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="dmlevents">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="I"/>
              <xsd:enumeration value="IU"/>
              <xsd:enumeration value="IUD"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="consumption">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="exclusive"/>
              <xsd:enumeration value="shared"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>

```

```

        <xsd:enumeration value="rule"/>
    </xsd:restriction>
</xsd:simpleType>
</xsd:attribute>
</xsd:restriction>
</xsd:complexContent>
</xsd:complexType>

<!-- Properties of a rule class with composite events -->
<xsd:complexType name="CompositeRuleSetProp">
    <xsd:sequence>
        <xsd:element name="object" type="rlmp:PrimEventProp" minOccurs="0"
            maxOccurs="unbounded"/>
    </xsd:sequence>

    <xsd:attribute name="ordering" type="xsd:string"/>
    <xsd:attribute name="storage" type="xsd:string"/>
    <xsd:attribute name="autocommit">
        <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                <xsd:enumeration value="yes"/>
                <xsd:enumeration value="no"/>
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="equal" type="xsd:string"/>
    <xsd:attribute name="consumption">
        <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                <xsd:enumeration value="exclusive"/>
                <xsd:enumeration value="shared"/>
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="duration">
        <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                <xsd:pattern value="call"/>
                <xsd:pattern value="([1-9]|[1-9][0-9]|[1-9][0-9]{2}|[1-9][0-9]{3})
                    (minutes|hours|days)"/>
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:attribute>
</xsd:complexType>

<!-- Primitive event properties with a composite event/rule class -->
<xsd:complexType name="PrimEventProp">
    <xsd:complexContent>
        <xsd:restriction base="xsd:anyType">
            <xsd:attribute name="type" type="xsd:string" use="required"/>
            <xsd:attribute name="consumption">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:string">
                        <xsd:enumeration value="exclusive"/>
                        <xsd:enumeration value="shared"/>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:attribute>
            <xsd:attribute name="duration">
                <xsd:simpleType>

```

```

        <xsd:restriction base="xsd:string">
            <xsd:pattern value="transaction"/>
            <xsd:pattern value="session"/>
            <xsd:pattern value="([1-9] | [1-9][0-9] | [1-9][0-9]{2} | [1-9][0-9]{3})
                (minutes|hours|days)"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:attribute>
</xsd:restriction>
</xsd:complexContent>
</xsd:complexType>
</xsd:schema>

```

Rule Condition

```

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xdb="http://xmlns.oracle.com/xdb"
    xmlns:rlmc="http://www.oracle.com/rlmgr/rulecond.xsd"
    elementFormDefault="qualified"
    targetNamespace="http://www.oracle.com/rlmgr/rulecond.xsd">

    <xsd:element name="condition">
        <xsd:complexType mixed="true">
            <xsd:choice>
                <xsd:element name="and" type="rlmc:AndType"
                    minOccurs="0" maxOccurs="1"/>
                <xsd:element name="any" type="rlmc:AnyType"
                    minOccurs="0" maxOccurs="1"/>
                <xsd:element name="object" type="rlmc:ObjectCondType"
                    minOccurs="0" maxOccurs="1"/>
            </xsd:choice>
        </xsd:complexType >
        <xsd:unique name="objNamesAny">
            <xsd:selector xpath="//object"/>
            <xsd:field xpath="@name"/>
        </xsd:unique>
    </xsd:element>

    <xsd:complexType name="AndType">
        <xsd:sequence>
            <xsd:element name="object" type="rlmc:ObjectCondType"
                minOccurs="1" maxOccurs="unbounded"/>
        <xsd:choice>
            <xsd:element name="not" type="rlmc:NotCondType"
                minOccurs="0" maxOccurs="1"/>

            <xsd:element name="notany" type="rlmc:NotAnyCondType"
                minOccurs="0" maxOccurs="1"/>
        </xsd:choice>
    </xsd:sequence>
    <xsd:attribute name="join" type="xsd:string"/>
    <xsd:attribute name="equal" type="xsd:string"/>
    <xsd:attribute name="sequence">
        <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                <xsd:enumeration value="yes"/>
                <xsd:enumeration value="no"/>
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:attribute>

```

```

    </xsd:attribute>
</xsd:complexType>

<xsd:complexType name="NotCondType">
  <xsd:sequence>
    <xsd:element name="object" type="rlmc:ObjectCondType"
      minOccurs="0" maxOccurs="1" />
  </xsd:sequence>
  <xsd:attribute name="by" type="xsd:string" />
  <xsd:attribute name="join" type="xsd:string" />
</xsd:complexType>

<xsd:complexType name="NotAnyCondType">
  <xsd:sequence>
    <xsd:element name="object" type="rlmc:ObjectCondType" minOccurs="1"
      maxOccurs="unbounded" />
  </xsd:sequence>
  <xsd:attribute name="count" type="xsd:positiveInteger" />
  <xsd:attribute name="by" type="xsd:string" />
  <xsd:attribute name="join" type="xsd:string" />
</xsd:complexType>

<xsd:complexType name="AnyType">
  <xsd:sequence>
    <xsd:element name="object" type="rlmc:ObjectCondType" minOccurs="1"
      maxOccurs="unbounded" />
  </xsd:sequence>
  <xsd:attribute name="count" type="xsd:positiveInteger" />
  <xsd:attribute name="join" type="xsd:string" />
  <xsd:attribute name="equal" type="xsd:string" />
  <xsd:attribute name="sequence">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="yes" />
        <xsd:enumeration value="no" />
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:attribute>
</xsd:complexType>
<xsd:complexType name="ObjectCondType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:string">
      <xsd:attribute name="name" type="xsd:string" use="required" />
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
</xsd:schema>

```

Implementing Various Forms of Rule Actions With the Action Callback Procedure

The action callback procedure configured for a rule class acts as a common entry point for executing the actions for all the rules in the rule class. This procedure is called once for each rule matching an event (primitive or composite). At the time of execution, this procedure has access to the events that matched the rule and the complete list of action preferences associated with the matched rule. The user implementing the action callback procedure can rely on this information to determine the appropriate action for each rule.

Rules Manager provides maximum flexibility by not restricting the types of action preferences used for a rule class. In the simplest case, the action preferences associated with a rule can be one or more scalar values that are used as arguments to a common procedure (OfferPromotion from the example in Section 2.4) that executes the appropriate action for each rule. Table G-1 represents one such rule class that is created with three types of action preference - PromoType, OfferedBy, and Discount.

Table G-1 *TravelPromotion Rule Class Table*

rlm\$ruleid	rlm\$rulecond	PromoType	OfferedBy	Discount	rlm\$ruledesc
AB_AV_ORL	Airline='Abcair' and ToCity='Orlando'	RentalCar	Acar	10	Additional info
AC_HT_SJC	Airline='Acbaair' and ToCity='San Jose'	Hotel	Ahotel	5	Additional info
...

If a single PL/SQL procedure in the database is implemented to offer all types of discounts by accepting appropriate arguments, the action callback procedure for the preceding rule class can make use of this procedure to execute appropriate actions, as follows:

```
CREATE or REPLACE PROCEDURE PromoAction (rlm$event  AddFlight,
                                           rlm$rule    TravelPromotion%ROWTYPE) is
BEGIN
    OfferPromotion(rlm$event.CustId,
                  rlm$rule.PromoType,
                  rlm$rule.OfferedBy,
                  rlm$rule.Discount);
END;
```

However, if the action for all the rules is not implemented with a single procedure but with a handful of procedures, the action callback procedure can be implemented to

choose the appropriate procedure using one of the action preference values. For example, if the procedures used to offer hotel and rental car promotions are different, the preceding action callback procedure can be implemented as follows:

```
CREATE or REPLACE PROCEDURE PromoAction (rlm$event AddFlight,
                                         rlm$rule TravelPromotion%ROWTYPE) is
BEGIN
  CASE rlm$rule.PromoType
    WHEN 'RentalCar' then
      OfferRentalCarPromotion(rlm$event.CustId,
                             rlm$rule.OfferedBy,
                             rlm$rule.Discount);
    WHEN 'Hotel' then
      OfferHotelPromotion (rlm$event.CustId,
                          rlm$rule.OfferedBy,
                          rlm$rule.Discount);
    ELSE
      OfferPromotion(rlm$event.CustId,
                    rlm$rule.PromoType,
                    rlm$rule.OfferedBy,
                    rlm$rule.Discount);
  END CASE;
END;
```

For complex rule applications requiring different actions for each rule, the PL/SQL commands that model the actions can be stored as the rule action preferences. For this purpose, the preceding rule class table can be configured to store the anonymous PL/SQL code blocks as the rule action preferences as described in [Table G-2](#).

Table G-2 Modified TravelPromotion Rule Class Table

rlm\$ruleid	rlm\$rulecond	ActionCommands	rlm\$ruledesc
AB_AV_ORL	Airline='Abcair' and ToCity='Orlando'	begin OfferAcarPromotion(:1,10); end;	Additional info
AC_HT_SJC	Airline='Acbaair' and ToCity='San Jose'	begin OfferAhotelPromotion (:1, 5); end;	Additional info
...

For the preceding rule class configuration, the action callback procedure can be implemented to execute the anonymous PL/SQL code blocks using the EXECUTE IMMEDIATE command shown as follows.

```
CREATE or REPLACE PROCEDURE PromoAction (rlm$event AddFlight,
                                         rlm$rule TravelPromotion%ROWTYPE) is
BEGIN
  EXECUTE IMMEDIATE rlm$rule.ActionCommands USING rlm$event.CustId;
END;
```

A rules application in the database can use a combination of the previous three procedures to model complex action execution logic. For this purpose, the rule class can be created with as high as 997 action preference columns, each with any valid SQL data type (including RAW, CLOB, and XML).

Index

A

action callback procedure, 2-3
 implementing various forms of rule actions, G-1
ADD_ELEMENTARY_ATTRIBUTE procedure, 10-5
ADD_FUNCTIONS procedure, 10-6
ALTER INDEX REBUILD statement, 15-4
ALTER INDEX RENAME TO statement, 15-5
AnyData.convertObject method, 10-10, 15-2
application examples, B-1
 active application, B-1
 batch evaluation, B-2
 law enforcement, 9-1
 order management, 9-8
 resource management, B-2
ASSIGN_ATTRIBUTE_SET procedure, 10-7, 14-2
attribute sets
 automatically creating, 10-5
 copying, 10-8
 creating with an existing object type, 10-5
 dropping, 10-6, 10-8
 examples, 10-5
 unassigning, 10-8
autocommit, 3-7, 3-8

B

BUILD_EXCEPTIONS_TABLE procedure, 10-11
bulk loading, 14-1
 bypassing validation, 14-1
bypassing validation, 14-1

C

complex rule conditions, 2-11
composite event, 2-1, 2-7
 rule conditions with ANY n semantics, 4-9
 rule conditions with set semantics, 4-9
 sequencing of primitive events, 4-4
conditional expressions
 See expressions
conflict resolution, 3-3
consumption of events, 3-1
converting rules applications, D-3
COPY_ATTRIBUTE_SET procedure, 10-8
CREATE INDEX statement, 11-6, 15-6

CREATE_ATTRIBUTE_SET procedure, 10-5

D

data item
 formatted as AnyData, 10-10
 formatted as name/value pair, 10-10
 formatted as string, 10-9
database
 exporting, 14-3
DBMS_EXPFIL package, 10-3
 GRANT_PRIVILEGE procedure, 10-11
 methods, 17-1
 REVOKE_PRIVILEGE procedure, 10-12
DBMS_RLMGR package
 methods, 7-1
decision points in you application, 1-4
DEFAULT_INDEX_PARAMETERS procedure, 11-6
DEFAULT_XPINDEX_PARAMETERS
 procedure, 12-5
DEFRAG_INDEX procedure, 11-8
deinstall script
 Rules Manager and Expression Filter, E-1
developing rules applications
 decision points, 1-4
 using Rules Manager, 1-4
DML events, 3-9
DROP INDEX statement, 15-9
DROP_ATTRIBUTE_SET procedure, 10-6, 10-8
duration of primitive events, 3-4

E

ECA components
 defined, 1-2
ECA rules, 2-4
 defined, 1-2
elementary attributes, 10-3
equality join predicates, 3-6
error messages, 10-12
EVALUATE operator, 10-9, 15-2
 arguments, 10-9
evaluating composite events
 using complex rule conditions, 2-11
evaluating rules
 incremental, 3-4

- negation in rules condition, 4-5
- event, 2-1
 - composite, 2-7
 - primitive, 2-5, 2-12, 5-1
- event management policies, 2-11, 3-1
 - autocommit, 3-7, 3-8
 - conflict resolution, 3-3
 - consumption of events, 3-1
 - DML events, 3-9
 - duration of primitive events, 3-4
 - equality join predicates, 3-6
 - order of rule execution, 3-4
 - specifying storage properties for objects created for the rule class, 3-7
- event structure, 2-1
- EXF\$ATTRIBUTE object type, 16-1
- EXF\$ATTRIBUTE_LIST object type, 16-1
- EXF\$INDEXOPER object type, 16-1
- EXF\$TABLE_ALIAS object type, 16-1
- EXF\$VALIDATE_n namespace, C-2
- EXF\$XPATH_TAG object type, 16-1
- EXF\$XPATH_TAGS object type, 16-1
- exporting
 - databases, 14-3
 - tables, 14-2
 - users, 14-3
- Expression column, 10-3, 10-7
 - creating, 10-3
- Expression data type, 10-3
 - creating a column of, 10-7
- Expression datatype, 10-7
- Expression Filter
 - active application example, B-1
 - batch evaluation example, B-2
 - configuring to process XPath predicates, 12-3
 - internal objects, C-1
 - overview, 10-1
 - resource management example, B-2
 - system triggers, C-2
 - usage scenarios, 10-1
 - utilities, 14-1
- expression sets, 10-3
 - allowing XPath predicates in, 12-1
- Expression Validation utility, 10-11
- expressions, 10-3
 - defined on one or more tables, A-1
 - definition, 10-3
 - deleting, 10-8
 - indexing, 11-1
 - inserting, 10-8
 - updating, 10-8
 - valid, 10-3
 - with spatial predicates, 13-1
 - with XPath predicates, 12-1

F

- features
 - new, xiii
- fragmentation of indexes, 11-8

Index-2

- functions
 - adding to attribute sets, 10-6

G

- GET_EXPRSET_STATS procedure, 11-7
- getVarchar methods
 - MEMBER, 10-10, 15-2
 - STATIC, 10-10, 15-2
- GRANT_PRIVILEGE procedure, 10-11

I

- importing
 - tables, 14-2
- incremental evaluation of rules, 3-4, 4-1
 - complex rule application
 - XML tag extensions, 4-1
- INDEX_PARAMETERS procedure, 11-6, 12-6
- indexed predicates, 11-4
- indexes
 - creating for expression set, 11-6, 15-6
 - creating from default parameters, 11-5
 - creating from exact parameters, 11-6
 - creating from statistics, 11-7
 - defragmenting, 11-8
 - dropping, 14-1, 15-9
 - maintaining, 11-8
 - processing, 11-3
 - processing for spatial predicates, 13-3
 - processing for XPath predicates, 12-4
 - rebuilding, 11-8, 15-4
 - storing, 11-8
 - tuning, 11-2, 11-5
 - tuning for XPath predicates, 12-5
 - usage, 11-7
- indexing, 11-1
 - and database objects, 11-8
 - predicates, 11-1
 - spatial predicates, 13-3
 - XPath predicates, 12-2
- INSERT_EXPRESSION privilege, 10-11
- INSERT privilege, 10-11
- install script
 - Rules Manager and Expression Filter, E-1
- installation
 - automatic of Rules Manager and Expression Filter, E-1
- internal objects, C-1

L

- loading expression data, 14-1

M

- matching rules
 - conflict resolution, 3-3
 - order of rule execution, 3-4
- metadata
 - expression set, 10-3

multitier mode
rule evaluation, 5-4

O

object types
AnyData.convertObject method, 10-10, 15-2
attribute set, C-1
event structure, 2-1
Expression Filter
EXF\$ATTRIBUTE, 16-1
EXF\$ATTRIBUTE_LIST, 16-1
EXF\$INDEXOPER, 16-1
EXF\$TABLE_ALIAS, 16-1
EXF\$XPATH_TAG, 16-1
EXF\$XPATH_TAGS, 16-1
Rules Manager
RLM\$EVENTIDS, 6-1
objects created for the rule class
specifying storage properties, 3-7
order of rule execution, 3-4

P

predicate operators, 11-2
predicate table, 11-2, 11-8
querying, 11-5
predicates
evaluating in a sparse predicate group, 11-5
evaluating in a stored predicate group, 11-5
evaluating in an indexed predicate group, 11-5
indexable, 11-1
indexed, 11-4
sparse, 11-3, 11-5
spatial, 3-12, 13-1
stored, 11-4
XPath, 12-1
primitive event, 2-1, 2-5, 2-12, 5-1
as an XML document, 3-10
defined as XML document, 3-11
primitive events using relational tables
storing in relational tables, 3-10
privileges
granting, 10-11
revoking, 10-11

R

results view, 2-3
REVOKE_PRIVILEGE procedure, 10-12
RLM\$EVENTIDS object type, 6-1
rule, 2-2
defined, 1-2
rule action
preferences, 2-2
rule class, 2-2
rule class properties
see event management policies
rule condition, 2-2, 4-1
ANY n semantics, 4-9
negation, 4-5

sequencing of primitive events, 4-4
set semantics, 4-9
with spatial predicates, 3-12
rule session, 2-3, 5-3
rules application
complex
XML tag extensions, 4-1
converting, D-3
creating
spanning multiple tiers, 2-12, 5-1
use composite event, 2-7
use simple or non-composite event, 2-5
event management policies, 2-11, 3-1
examples
law enforcement, 9-1
order management, 9-8
rule evaluation
multitier mode, 5-4
single tier mode, 5-4

rules class
process rules defined on XML documents, 3-10, 3-11

Rules Manager
developing rules applications, 1-4
five elements of an application, 1-2
internal objects, C-1
overview and introduction, 1-1
process steps
creating and implementing rules
application, 1-3
use cases, 9-1
rules session, 5-1

S

secondary objects, 11-8
SELECT privileges, 10-11
single tier mode
rule evaluation, 5-4
sparse predicates, 11-5
spatial predicates, 3-12, 13-1
indexable, 13-3
using in expressions, 13-1
SQL*Loader, 14-1
stored predicates, 11-4
system triggers, C-2

T

table alias, A-1
attributes, 3-9, 3-10
tables
exporting, 14-2
importing, 14-2
triggers, C-2
system, C-2
validation, C-2

U

UNASSIGN_ATTRIBUTE_SET procedure, 10-8, 14-1

- UPDATE EXPRESSION privilege, 10-11
- UPDATE privilege, 10-11
- USER_EXPFIL_ASET_FUNCTIONS view, 18-1
- USER_EXPFIL_ATTRIBUTE_SETS view, 18-2
- USER_EXPFIL_ATTRIBUTES view, 18-2
- USER_EXPFIL_DEF_INDEX_PARAMS view, 18-2
- USER_EXPFIL_EXPRESSION_SETS view, 18-3
- USER_EXPFIL_EXPRSET_STATS view, 18-3
- USER_EXPFIL_INDEX_PARAMS view, 18-4
- USER_EXPFIL_INDEXES view, 18-4
- USER_EXPFIL_PREDTAB_ATTRIBUTES view, 18-5
- USER_EXPFIL_PRIVILEGES view, 18-5
- USER_RLMGR_COMPRCLS_PROPERTIES view, 8-3
- USER_RLMGR_COMPRSET_PROPERTIES view, 8-3
- USER_RLMGR_EVENT_STRUCTS view, 8-1
- USER_RLMGR_PRIVILEGES view, 8-2
- USER_RLMGR_RULE_CLASS_STATUS view, 8-2
- USER_RLMGR_RULE_CLASSES view, 8-1
- users
 - exporting, 14-3

V

- VALIDATE_EXPRESSIONS procedure, 10-11, 14-2
- validation semantics, 10-11
- validation trigger, C-2

X

- XML schema
 - rule class properties, F-1
 - rule condition, F-3
- XML tags
 - extensions, 4-2
- XMLType datatype, 12-1
- XPath predicates, 12-1
 - configuring Expression Filter for, 12-3
 - index tuning for, 12-5
 - indexable, 12-3
 - indexing set of, 12-2