

Oracle® Database

Rules Manager and Expression Filter Developer's Guide

11g Release 2 (11.2)

E14919-04

August 2011

Obsolescence Notice - see My Oracle Support Note ID
1244535.1

Oracle Database Rules Manager and Expression Filter Developer's Guide, 11g Release 2 (11.2)

E14919-04

Copyright © 2003, 2011, Oracle and/or its affiliates. All rights reserved.

Primary Authors: Aravind Yalamanchi, Rod Ward

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

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. 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, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

Preface	xi
Audience	xi
Documentation Accessibility	xi
Related Documents	xi
Conventions	xii
What's New in Rules Manager and Expression Filter?	xiii
1 Introduction to Rules Manager	
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	2-9
2.4.2 Evaluating Composite Events Using Complex Rule Conditions	2-12
2.5 Setting Event Management Policies (Rule Class Properties) for Rule Applications	2-12
2.6 Creating Rules Applications That Span Multiple Tiers	2-13
2.7 Using Rules Manager with SQL*Loader and Export/Import Utilities	2-13
2.7.1 SQL*Loader	2-13
2.7.2 Export/Import	2-14
3 Event Management Policies	
3.1 Consumption of Events	3-2
3.2 Ordering of Rule Execution	3-3
3.3 Duration of Events	3-5
3.4 Equality	3-6
3.4.1 Single Equal Specification for a Rule Class	3-7
3.4.2 Alternate Equal Specifications	3-8

3.5	Storage Properties	3-9
3.6	AUTOCOMMIT	3-10
3.7	DML and CNF Events	3-11
3.8	Rule Class Property Dependencies and Defaults.....	3-11

4 Event and Rule Class Configurations

4.1	Rules Specified on Relational Tables.....	4-1
4.2	Rule Conditions For XML Events.....	4-2
4.3	Rule Conditions with Spatial Predicates	4-3
4.4	Rule Conditions for Text Events.....	4-4
4.5	Disabling and Enabling Rules	4-5
4.6	Shareable Primitive Rule Conditions	4-5
4.7	Events Through Continuous Query Notification (Formerly Database Change Notification). 4-8	
4.8	Collection Events.....	4-9
4.8.1	Setting a Fixed Window Size or Length to Restrict the Number of Primitive Events..... 4-10	
4.8.2	Using Collection Primitive Events to Form Composite Events	4-10
4.8.3	Collection Events Have NULL Values Stored in its rlm\$CrtTime (Timestamp) Attribute 4-12	
4.9	Performance Tuning	4-13
4.10	Database State in Rule Conditions	4-14
4.11	Resetting Events for Development Environments.....	4-14

5 Rule Conditions

5.1	Support for Incremental Evaluation of Rules	5-1
5.2	Rule Conditions with Sequencing	5-5
5.3	Rule Conditions with Negation	5-6
5.4	Rule Conditions with Set Semantics	5-9
5.5	Rule Conditions with Any n Semantics.....	5-10
5.6	Rule Conditions with Collection Events.....	5-13
5.6.1	Setting a Fixed Window Size or Length to Restrict the Number of Primitive Events..... 5-14	
5.6.2	More About Using the Having Clause	5-16

6 Rules Applications That Span Multiple Tiers

6.1	Creating Rules Applications That Span Multiple Tiers.....	6-1
6.2	Modes of Operation.....	6-4
6.2.1	Single Tier Mode	6-4
6.2.2	Multitier Mode	6-4
6.2.2.1	Actions in the Mid-Tier.....	6-5

7 Rules Manager Object Types

8 DBMS_RLMGR Package

9 Rules Manager Views

9.1	USER_RLMGR_EVENT_STRUCTS View	9-1
9.2	USER_RLMGR_RULE_CLASSES View	9-1
9.3	USER_RLMGR_RULE_CLASS_STATUS View.....	9-2
9.4	USER_RLMGR_PRIVILEGES View	9-2
9.5	USER_RLMGR_COMPRCLS_PROPERTIES View.....	9-3

10 Rules Manager Use Cases

10.1	Law Enforcement Rules Application	10-1
10.2	Order Management Rules Application.....	10-8
10.3	Use of Collections in an Order Management Application.....	10-15

Part II Expression Filter

11 Oracle Expression Filter Concepts

11.1	What Is Expression Filter?	11-1
11.1.1	Expression Filter Usage Scenarios.....	11-1
11.2	Introduction to Expressions	11-3
11.2.1	Defining Attribute Sets	11-4
11.2.2	Defining Expression Columns	11-6
11.2.3	Inserting, Updating, and Deleting Expressions	11-8
11.3	Applying the SQL EVALUATE Operator	11-9
11.4	Evaluation Semantics	11-10
11.5	Granting and Revoking Privileges	11-11
11.6	Error Messages	11-12

12 Indexing Expressions

12.1	Concepts of Indexing Expressions.....	12-1
12.2	Indexable Predicates.....	12-1
12.3	Index Representation.....	12-2
12.4	Index Processing	12-3
12.5	Predicate Table Query.....	12-5
12.6	Index Creation and Tuning	12-5
12.7	Index Usage	12-7
12.8	Index Storage and Maintenance	12-8

13 Expressions with XPath Predicates

13.1	Using XPath Predicates in Expressions	13-1
13.2	Indexing XPath Predicates.....	13-2
13.2.1	Indexable XPath Predicates	13-3

13.2.2	Index Representation	13-4
13.2.3	Index Processing	13-4
13.2.4	Index Tuning for XPath Predicates	13-5
14	Expressions with Spatial and Text Predicates	
14.1	Expressions with Spatial Predicates.....	14-1
14.1.1	Using Spatial Predicates in Expressions.....	14-2
14.1.2	Indexing Spatial Predicates	14-3
14.2	Expressions with Text Predicates	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	Using Expression Filter with Utilities	
16.1	Bulk Loading of Expression Data	16-1
16.2	Exporting and Importing Tables, Users, and Databases.....	16-2
16.2.1	Exporting and Importing Tables Containing Expression Columns.....	16-2
16.2.2	Exporting a User Owning Attribute Sets	16-3
16.2.3	Exporting a Database Containing Attribute Sets	16-3
17	Object Types	
18	Management Procedures Using the DBMS_EXPFIL Package	
19	Expression Filter Views	
19.1	USER_EXPFIL_ASET_FUNCTIONS View	19-1
19.2	USER_EXPFIL_ATTRIBUTES View.....	19-2
19.3	USER_EXPFIL_ATTRIBUTE_SETS View.....	19-2
19.4	USER_EXPFIL_DEF_INDEX_PARAMS View	19-2
19.5	USER_EXPFIL_EXPRESSION_SETS View.....	19-3
19.6	USER_EXPFIL_EXPRSET_STATS View	19-3
19.7	USER_EXPFIL_INDEX_PARAMS View	19-4
19.8	USER_EXPFIL_INDEXES View	19-4
19.9	USER_EXPFIL_PREDTAB_ATTRIBUTES View	19-5
19.10	USER_EXPFIL_PRIVILEGES View	19-5
19.11	USER_EXPFIL_TEXT_INDEX_ERRORS.....	19-6

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

11-1	Defining an Attribute Set From an Existing Object Type	11-5
11-2	Defining an Attribute Set Incrementally	11-5
11-3	Adding User-Defined Functions to an Attribute Set	11-6
11-4	Inserting an Expression into the Consumer Table	11-9
11-5	Inserting an Expression That References a User-Defined Function	11-9

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
5-1	Hierarchical View of the XML Tag Extensions.....	5-3
11-1	Expression Filter Implementation Process for a Rules Application.....	11-4
11-2	Expression Data Type.....	11-8
12-1	Conceptual Predicate Table	12-3
13-1	Conceptual Predicate Table with XPath Predicates.....	13-4
14-1	Text Predicate in the Stored Expression Using the CONTAINS Operator	14-5

List of Tables

3-1	Valid and Invalid Rule Class Property Combinations	3-12
5-1	Relational View of the XML Tag Extensions.....	5-4
7-1	Rules Manager Object Types.....	7-1
8-1	DBMS_RLMGR Procedures.....	8-1
9-1	Rules Manager Views.....	9-1
9-2	USER_RLMGR_EVENT_STRUCTS View	9-1
9-3	USER_RLMGR_RULE_CLASSES View	9-2
9-4	USER_RLMGR_RULE_CLASS_STATUS View.....	9-2
9-5	USER_RLMGR_PRIVILEGES View	9-3
9-6	USER_RLMGR_COMPRCLS_PROPERTIES View	9-3
15-1	Expression Filter Index Creation and Usage Statements	15-1
17-1	Expression Filter Object Types.....	17-1
18-1	DBMS_EXPFIL Procedures.....	18-1
19-1	Expression Filter Views.....	19-1
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 Rules Manager and Expression Filter Developer's Guide 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 Rules Manager and Expression Filter Developer's Guide 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

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc>.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit

<http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info> or visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs> if you are hearing impaired.

Related Documents

For more information, see these Oracle resources:

- *Oracle Database SQL Language Reference*
- *Oracle Database Utilities*

- *Oracle Database Error Messages*
- *Oracle Database Performance Tuning Guide*
- *Oracle XML DB Developer's Guide*
- *Oracle Database Advanced Application Developer's Guide*
- *Oracle Database Object-Relational Developer's Guide*
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Text Application Developer's Guide*

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 11g Release 2 (11.2) and provides pointers to additional information.

There are no new features for Rules Manager and Expression Filter in Oracle Database 11g Release 2 (11.2).

Obsolescence Notice: Rules Manager and Expression Filter features of Oracle Database are obsolete in the next major release after Oracle Database 11g Release 2 (11.2). Support will only be available for the life of Oracle Database Release 11g Release 2. See My Oracle Support Note ID 1244535.1 for more information.

Introduction to Rules Manager

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. When you manage rules in Oracle Database, rules keep pace with changing business conditions and are always up-to-date; using SQL, you can easily change rules because rules are not hard-coded in your application or loaded into a memory-based rules repository. You can efficiently evaluate rules when you store the complete business context in your Oracle Database as well as provide data 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, as a database feature, you can use Rules Manager 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 Rules Manager performs a prescribed rule action, 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.

[Section 1.1](#) briefly introduces the concept of rules 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 you specify using the attributes you define in a corresponding event structure. When an instance of the event structure satisfies a rule condition, a rule action takes place. When the rule action executes, event management policies define how to handle an event instance. 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 it performs some prescribed action. 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 clause 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 you define as an object type with attributes that describe specific features of an event.
2. A rule consists of a condition and action preferences.

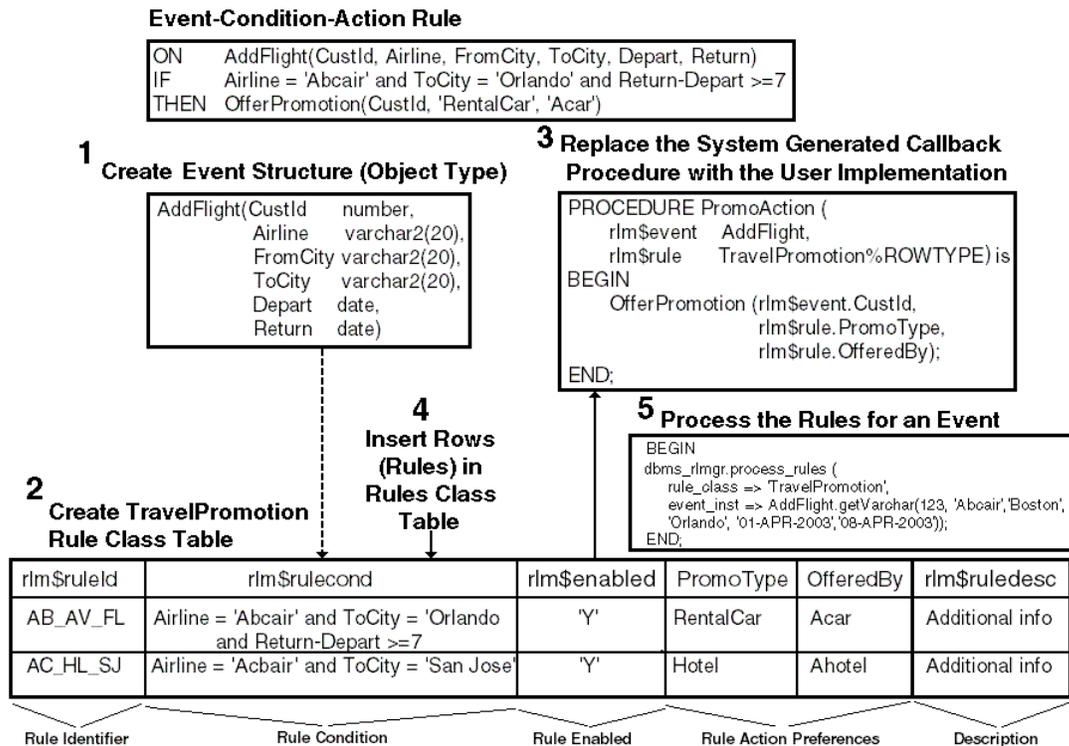
- A rule condition you express using the attributes defined in the event structure.
 - Rule action preferences to determine the exact action for each rule and to specify the details for the action.
3. A rule class that is a database table that stores and groups the rules defined for an event structure.
 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, sequencing, and collections. Rules Manager supports incremental evaluation of rules involving composite events. You can specify complex rule conditions using XML tags within conditional expressions in the SQL WHERE clause format. You can enforce rule class event management policies, such as consumption, conflict resolution, and duration, for each rule application. Figure 1–1 shows the process steps for creating and implementing a Rules Manager 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 10.

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 does it execute 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 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 you use 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 5, "Rule Conditions"](#)
- [Chapter 6, "Rules Applications That Span Multiple Tiers"](#)
- [Chapter 7, "Rules Manager Object Types"](#)
- [Chapter 8, "DBMS_RLMGR Package"](#)
- [Chapter 9, "Rules Manager Views"](#)
- [Chapter 10, "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 you define 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. So Rules Manager can fully evaluate the rules defined for the corresponding event structure 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 Rules Manager cannot fully evaluate the rules defined for the composite event structure 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** is a conditional expression formed using the attributes defined in the event structure. For example, the following rule condition use the attributes: `Airline`, `ToCity`, `Return`, and `Depart`:

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

- The **rule action preferences** 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 you use a rule class 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 determine the appropriate action for the rule.

Action preferences can come in different forms, such as:

- * A list of literals 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. Thus, the component issuing the query executes the corresponding actions. This is useful when the action for certain rules is implemented in the application on multiple tiers. See [Section 2.6](#) for more information.

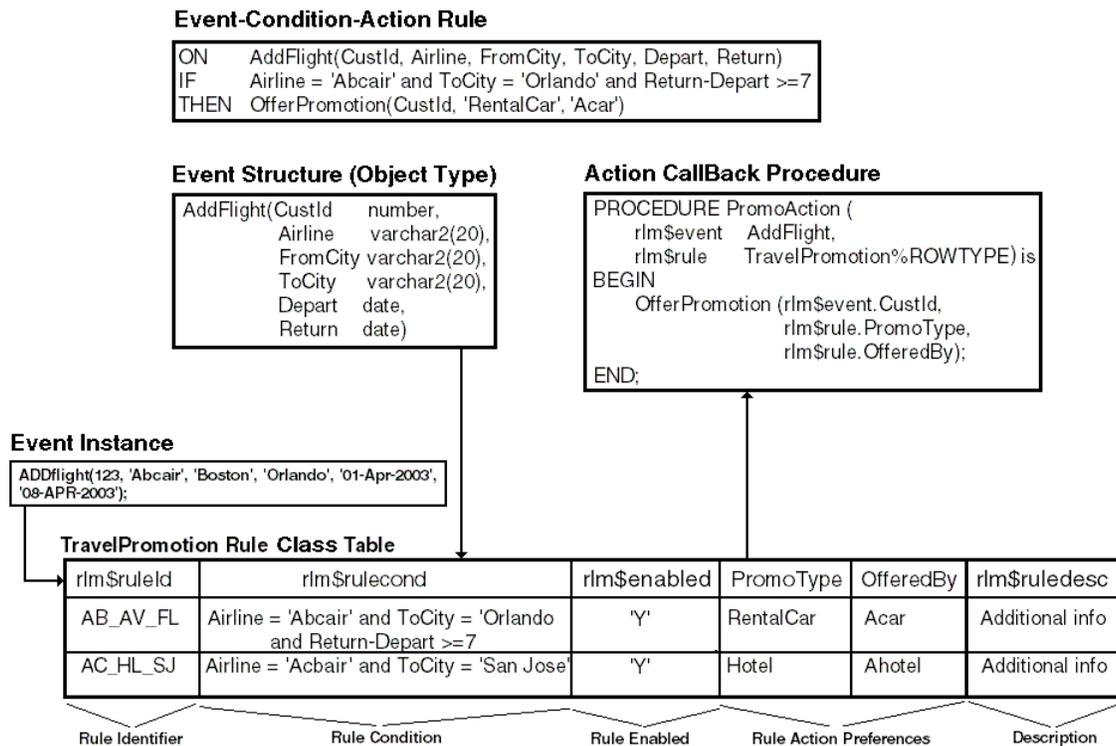
- The results from a rule evaluation are available through the results view until the end of the **rule session**. By default, the database session (from connect to disconnect) is considered the rule session. Alternatively, you can use the reset session procedure (`dbms_rlmgr.reset_session()`) to end 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.
- `rlm$enabled` — contains a value indicating whether the rule added to the rule class is enabled or disabled. A value of `Y` indicates the rule is enabled, a value of `N` indicates that it is disabled. By default, a rule created with a missing value for the `rlm$enabled` column is considered enabled.
- `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 `PromoType` column stores the type of promotion to be offered, such as a car rental promotion or hotel stay promotion. 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.

Each row of the `TravelPromotion` rule class table stores an ECA rule. 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 you add all the rules 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$enabled     CHAR(1) DEFAULT 'Y',
    rlm$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;
    
```

Rule class creation, in this case, creates the action callback procedure 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. Insert the following rule 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 11.3](#). 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 5](#) 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 5.1](#) for more information.

Note: The intermediate state maintained for a rule is closely related to the corresponding rule condition (for composite events). So, any modifications made to the rule condition (using the `UPDATE` command) will discard the intermediate state associated with the rule and the state is maintained only for the events processed subsequently. Effectively, updating a rule condition is equivalent to deleting the corresponding rule and inserting a new one. Modifying the rule's action preferences or the rule identifier has no impact on the rule evaluation state.

- 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 5](#) for more information.

- Setting event management policies

Rules Manager allows an individual with application domain knowledge to declaratively set event management policies for a rules application. Event policies are specified as properties of a rules class when the rule class is created to control the behavior of simple and composite events in the system, and the performance of composite events.

The policies controlling the ordering of rule executions and the reuse of events for multiple rule executions are applicable to an application with simple as well as composite events. Other composite event-specific policies control the aging of the unused events, ordering of events, and the correlation of primitive events within composite events. The event management policies are summarized in [Section 2.5](#) and described in [Section 3.1](#) through [Section 3.8](#).

Note: The `EQUAL` property must be specified for a rules class if it is configured for composite events. Domain knowledge is needed to identify common equality join predicates that correlate the primitive events for all the rules in a rule class.

Designing Rules Applications with Composite Events

Developing a rules application for composite events has some similarities with that of developing a database (SQL) application. The event structure definitions in a rules application are similar to table definitions in a database application. SQL queries operating on these tables are similar to the rule conditions defined in a rule class. In a database application, constraints and indexes specific to each application are created for data integrity and performance. Similarly, in the case of a rules application, properties specified for the rule class enforce the event management policies and improve the performance. These rule class properties are summarized in [Section 2.5](#) and described [Chapter 3](#).

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

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

Use the attribute names, `Flt` and `Car`, 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. Configure the rule class 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 equal="Flt.CustId, Car.CustId"/>');
END;
```

The previous code example creates the rule class for the composite event structure. The `rlcls_prop` argument specifies the XML element `<composite>` to configure the rule class for composite events. The properties also include an `equal` specification that identifies the common equality join predicate in all the rules in the rule class. Other critical rule class properties such as consumption, duration, and ordering of events can be specified using the syntax discussed in [Section 3.1](#) through [Section 3.7](#).

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. The action callback procedure includes additional arguments when the rule class is configured for the `RULE` consumption policy or when the rule class is enabled for one or more collection events.

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 5.1](#) 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

Rules Manager supports evaluating composite events using complex rule conditions 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 adds special semantics to these expressions. [Chapter 5](#) describes more about each type of complex rule condition. [Section 5.1](#) 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 rules 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`, `UPDATE`, `DELETE`) on the corresponding tables as the events for which the rules are evaluated
- CNF Events — continuous query notification (formerly database change notification) events similar to DML Events except that the rules are processed after the commit of the transaction performing the DML operations.

You specify the rule class properties 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, specify these properties 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 3.8](#) 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 6](#) 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 6.1](#) 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 6.1](#) for the details.
6. Find the matching rules by querying the results view. See Step 6 in [Section 6.1](#) for the details.
7. Consume the event that is used in a rule execution. See Step 7 in [Section 6.1](#) for the details.

For more information about creating rules applications that span multiple tiers, see [Section 6.1](#), and for more information about running rules applications in multitier mode see [Section 6.2](#).

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

You can use SQL*Loader to bulk load data from an ASCII file into a rule class table. For the loader operations, SQL*Loader treats the rule conditions stored in the `rlm$rulecond` column of the rule class table 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. The values for the action preference columns in the rule class table are loaded using normal SQL*Loader semantics.

Rules Manager automatically validates the data loaded into the rule condition column 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

You can export a rules application defined using a set of event structures and a rule class and then import it 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 you export a rule class, 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 you use the `tables` parameter 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 operation.

Note: In the case of a rule class with references to shareable primitive rule conditions, the conditions table storing the conditions are not exported unless the schema is exported or the conditions table is explicitly listed in the `tables` parameter of the export command. See the note at the end of [Section 4.6](#) for more information.

You can use the dump file, created with the export of a rule class, 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.

Note: In the case of the rule class with references to shareable primitive rule conditions, the rules are validated during the import operation. Any broken references found due to a missing conditions table or the specific primitive condition in the conditions table, returns an ORA-41704 error message. However, the broken references can be fixed as a post-import operation. For this purpose, all the rule conditions with broken references are marked FAILED with the corresponding `rlm$enabled` column storing the value F.

Event Management Policies

The rule class properties you specify at the time you create the rule class 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. Rules Manager supports application-specific requirements for reusing primitive events and for handling duplicate composite events using rule event management policies 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 you create an event structure with one or more table alias attributes, that you can configure the corresponding rule class to consider the DML operation (`INSERT`, `UPDATE`, `DELETE`) on the corresponding tables as the events for which the rules are evaluated. The DML Events specification uses DML events from uncommitted transactions to process the rules. Whereas, in the case of continuous query notification (formerly database change notification) (CNF) events (see [Section 2.5](#) and [Section 3.7](#) for more information), the rules are processed for the DML operations within a transaction after the transaction commit operation.

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. In addition to the event management policies, the rule class properties allow the specifications for collection of events. A collection specification enables a primitive event to be used in rule conditions involving collections of events as opposed to individual events. For such events, rule conditions can compute aggregate values over

a finite but potentially large number of primitive events of the same type and specify predicates on the resulting aggregates. Primitive events of a specific type are grouped based on certain event attributes and aggregate operators, such as `SUM`, `AVG`, `MIN`, `MAX`, and `COUNT` on the other event attributes are used to apply predicates.

The rule class properties are all specified in an XML properties document, which is used as one of the arguments (`rlcls_prop`) to the rule class creation procedure (`dbms_rlmgr.create_rule_class`). [Section 4.8](#) discusses the rule class property for enabling collections. All other rule class properties are 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, you can combine two instances of the `AddFlight` event with one instance of `AddRentalCar` event to form two different composite events (that could match two different rules). Some rule applications require 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 3.2](#) that describes ordering). You can configure the rule class created for a primitive event structure 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 you configure the rule class 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. You use these identifiers to selectively consume some or all of the primitive events atomically. See the `DBMS_RLMGR.CONSUME_PRIM_EVENTS` function for more information.

3.2 Ordering of Rule Execution

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 information later in this section 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, you can use the rule identifiers and the action preferences associated with the rules 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$ruleid`. As shown in this example, the rule identifier `rlm$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, you can use the `DESC` keyword with some of the attributes to sort the rules in the descending order (see information later in this section for complete syntax). When you configure the rule class 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 using the `ORDERING` property.

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, you can specify the ordering of the resulting events and the matching rules using the attributes in the events, the action preferences associated with the rules, and the rule identifiers. For the travel services rule class example, you can specify the ordering of the events and the matching rules 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, you use the rule identifier `rlm$rule` to refer to the attributes associated with the rule class (action preferences `PromoType` and `OfferedBy` and the rule identifier `rlm$ruleid`); and you use the variables declared for the primitive events in the composite event structure (`Flt` for `AddFlight` and `Car` for `AddRentalCar`) 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 Rules Manager only allows in the ordering clause, the rule identifier and action preferences associated with the rules.

3.3 Duration of Events

It is common for applications to generate events that will never trigger a rule firing, thus never consuming these events. The duration policy for primitive events determines the maximum lifetime of the events. When you add a primitive event to 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 Rules Manager detects all the required primitive events 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 Rules Manager detects all the required primitive events 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. You can configure a subset of primitive events within a composite event 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. You specify the event timeout 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="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, you can specify them 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, you use the value of 10 days for the `duration` attribute of the `<composite>` element 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.

3.4 Equality

In the case of a rule class for composite events, identifying the most-common equality join predicates in all rule conditions and specifying those using the `EQUAL` rule class property is important for performance. All rules in a rule class use one or more common (equality) join predicates to relate the primitive events that form a composite event with each other. You define these join predicates 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`). You can configure the rule class to optimize a limited number of distinct equality join predicates used in its rule conditions using the `EQUAL` property.

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

The `EQUAL` property used for a rule class may be of two types depending on the homogeneity of the join conditions in its rules. In the case of a rule class with a homogenous set of rules using the same equality join predicate for all its rule conditions, the equal specification for the rule class can be uniquely identified and there is only a single equal specification. On the other hand, when different subsets of rules use different join predicates, you can configure the rule class with a limited number of alternate equal specifications. Each equal specification may be based on a

single attribute from each contributing primitive event or it could be based on multiple attributes from each primitive event (concatenated keys) as discussed in this section.

3.4.1 Single Equal Specification for a Rule Class

A single equal specification for a rule class identifies the equality join predicates that are used by all rules in the rule class. For example, if all the rules in the travel services applications relate the primitive events based on the customer identifiers (`Flt.CustId = Car.CustId`), then you can configure this join predicate as a single equal specification for the rule class's EQUAL property.

In this case, you specify the EQUAL property 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 composite event has more than two primitive events, the corresponding rule conditions may employ conjunctions of two or more equality join predicates. For example, if `reading1`, `reading2`, and `reading3` are three primitive events representing RFID readings, the join condition in a rule relating these three events could be `reading1.ReaderId = reading2.ReaderId` and `reading2.ReaderId = reading3.readingId` (to check for all three readings to occur at the same reader). The corresponding equal specification is a comma-delimited list of attributes from each primitive event (`reading1.readerId, reading2.readerId, reading3.readerId`).

In the case of single equal specification, since each rule condition is guaranteed to use the same equality join predicate, the equal specification in the rule class properties obviate the need for the same join predicate in each rule condition. Hence, the rules in the rule class may skip the equality join predicates involving the same set of attributes, as demonstrated with the following examples.

The following rule condition explicitly specifies the equality join predicate in conjunction with other (possible inequality) join predicates. This specification uses the SQL WHERE clause syntax for join predicates.

```
<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>
```

The following rule condition demonstrates the use of the EQUAL clause in the place of the equality join predicate in the previous example. The EQUAL clause specification for a rule condition acts as a short representation of equality join predicates especially when the rule condition has negation (see [Section 5.3](#)) or Any *n* (see [Section 5.5](#)) constructs.

```
<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>
```

When the EQUAL property for the rule class is specified as `equal="Flt.CustId, Car.CustId"`, use of the corresponding join predicate or the EQUAL clause in a rule condition (as shown with the previous two examples) is redundant. In this case, the single equal specification associated with the rule class is enforced for all rules in the rule class. Hence, the following rule condition is equivalent to the previous two examples when the rule class is created with the previous EQUAL property.

```
<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>
```

Equal Specification with Concatenated Keys

Often the equality join predicates between primitive events may involve more than one attribute from each primitive event. For example, in the travel services application, the `AddFlight` and `AddRentalCar` events may be related to each other based on their itineraries (equality predicate on `Depart` and `CheckOut` dates from respective events), in addition to the equality of the customer identifiers. A sample rule using such join predicates is as follows.

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

If the equality predicates involving multiple attributes from each primitive event are common across all the rules in the rule class, you can configure the rule class with an EQUAL property specification with concatenated keys for optimal performance.

```
<composite equal="(Flt.CustId, Car.CustId), (Flt.Depart, Car.CheckOut)"/>
```

Note: A maximum of three key concatenations can be specified with the rule class's EQUAL property.

With the previous specification, the combination of `[Flt.CustId, Flt.Depart]` attributes acts as a concatenated key for each `Flt` event and it should match the concatenated key from a `Car` event for any rule in the rule class to be true. Since you enforce the previous equal specification for all rules in the rule class, a similar equal specification for each rule may be skipped.

3.4.2 Alternate Equal Specifications

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 single equal specification is grouped using parentheses and alternate equal specifications are separated using a vertical bar (`'|'`) character. For example, if you create a rule class 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`). You can optimize the rule class to process both types of these rules efficiently using the following EQUAL property:

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

Note: At most, you can define five alternate equal specifications for a rule class's EQUAL property.

The alternate equal specifications provide a means for optimizing the rule evaluation for the most common join predicates in a rule class and the rule class does not automatically enforce any equality join predicates for its rules. For optimal performance, each rule condition in the rule class must specify one of the alternate equal specifications for its EQUAL clause. For example, the following rule's EQUAL clause matches one of the alternate equal 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 clause 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.

The alternate equal specification may include one or more specifications involving concatenated keys. For example, if the travel services application uses some rules, which just relate the AddFlight and AddRentalCar events based on their customer identifiers and some other rules on the identifiers as well as the dates in their itineraries, you can configure the rule class with the following equal property for optimal performance.

```
<composite equal="(Flt.CustId, Car.CustId) |
                (Flt.CustId, Car.CustId), (Flt.Depart, Car.CheckOut)"/>
```

With this specification at the rule class level, individual rules in the rule class may use either of these two alternate EQUAL property specifications.

```
<condition>
  <and equal="(Flt.CustId, Car.CustId), (Flt.Depart, Car.CheckOut)">
    <object name="Flt"> Airline='Abcair' and ToCity='Orlando' </object>
    <object> CarType = 'Luxury' </object>
  </and>
</condition>
```

Note that while matching an EQUAL clause specified for a rule with one of the alternate equal specifications, the order of attributes is irrelevant.

3.5 Storage Properties

Use the STORAGE attribute of the <simple> or <composite> element 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 you can specify in a typical SQL CREATE TABLE statement. The following XML

properties document shows how to use it (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"/>
```

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

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

3.6 AUTOCOMMIT

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

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

You can specify the `AUTOCOMMIT` property 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, you should perform any DDL operations 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 you create the rule class 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 you define the event structure with one or more table alias constructs and you configure the corresponding rule class for DML events (see [Section 3.7](#)), 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 5.3](#)).

3.7 DML and CNF Events

When you create an event structure with one or more table alias attributes (see [Section 4.1](#)), then you can configure the corresponding rule class to consider the SQL `INSERT` and `SQL*Loader` operations on the corresponding tables as the events for which the rules are evaluated. You can enable this rule class behavior using the `DMLEVENTS` or `CNFEVENTS` property for the rule class:

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

You can specify either of these properties for a rule class configured for simple and composite events. Events for `UPDATE` and `DELETE` operations on the underlying tables are only applicable for the rule classes configured for composite events.

```
<composite dmlevents="IUD"/>
<composite cnfevents="IUD"/>
```

When a row in a table is deleted, the state information for the rules matching this row are marked for deletion. Similarly, when the row is updated, the existing state information is marked for deletion and the new state information is computed for the updated row. The deleted row (or the old image of the updated row) does not have an effect on the past rule states. That is, the delete operation does not cause existing rule states to automatically become true due to the retraction of the event. This scenario pertains to the rule conditions with negative constructs, in which the event matched the negative portion of the rule before it is deleted.

With the `DMLEVENTS` specification, Rules Manager uses the events generated from a DML operation to process the rules in the rule class as part of the same DML command. This is achieved with the use of row level triggers on the underlying tables. On the other hand, when you use the `CNFEVENTS` specification, the rules are processed after the commit of the DML transaction using the net data changes (within the transaction) as the events. In effect, if a row is inserted into a table and then updated within the same transaction, with the `CNFEVENTS` specification, the rules are processed once for the newly inserted row (with committed data). Whereas, when you use the `DMLEVENTS` specification, the rules are processed twice for the same row - once synchronously with the `INSERT` operation and again with the `UPDATE` operation. See [Section 4.7](#) for additional information on the use of `CNFEVENTS`.

When you specify the `DMLEVENTS` policy, 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.

3.8 Rule Class Property Dependencies and Defaults

You can mix and match most of the rule class properties (or event management policies) described in this section while defining a rule class. However, [Table 3-1](#) shows some of the combinations of these properties that are considered invalid. 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 = IUD                               : NO
  ELSE IF DURATION = TRANSACTION / SESSION         : NO
  ELSE                                               : YES
```

Event and Rule Class Configurations

For a rule application, the types of data you include in an event structure and the event sources are application dependent. Rules Manager makes use of the Oracle rich type system to support rules applications involving complex data types such as XML, Spatial, and Text. Similarly, it leverages from its integration with the database to capture transactional and non-transaction modifications to the data as the source of events.

This chapter discusses various event and rule class configurations that provide the ultimate flexibility in designing rules applications in the database.

4.1 Rules Specified on Relational Tables

The rule applications considered so far use rule conditions that you define 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, you can use the row identifiers (ROWIDs) of these rows to pass the data by reference to the Rules Manager procedures, for example, using the `event_inst` parameter to represent an event instance of the `PROCESS_RULES` call (`event_inst => :FlightDataRowid`). For this purpose, you should model the corresponding event structure using Expression Filter's table alias constructs. See the `ADD_ELEMENTARY_ATTRIBUTE` procedure for more information. See [Section 10.2](#) and [Appendix A](#) for more examples.

For the travel services application considered in [Section 2.4](#), if you store the `AddFlight` and `AddRentalCar` primitive events in two relational tables `FlightData` and `RentalCarData` respectively, you can create the corresponding composite event structure 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
    tab_alias => rlm$table_alias('FlightData'));
  DBMS_RLMGR.ADD_ELEMENTARY_ATTRIBUTE (
    event_structure => 'TSCompEvent',
    attr_name => 'Car', --- Prim event name
    tab_alias => rlm$table_alias('RentalCarData'));
END;
```

Now you can use the composite event structure `TSCompEvent` 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 you use these ROWIDs 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 you evaluate the conditions corresponding to a primitive event 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, you can eliminate the `PROCESS_RULES` call by configuring the preceding rule class to consider all the DML (INSERT, UPDATE, and DELETE) operations on the corresponding tables as events. This is done using the `DMLEVENTS` or `CNFEVENTS` property at the time of rule class creation. (See [Section 3.7](#) and [Section 3.8](#)).

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 5.2](#)). This is because the implicit attribute `rlm$crtime` may not exist for the rows stored in the relational tables. You can enforce partial sequencing using the `join` property in the rule conditions.

4.2 Rule Conditions For XML Events

You can use the `XMLType` data type supplied by Oracle Database 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`. You specify the predicates on the `XMLType` attributes using the `EXTRACT` and `EXISTSNODE` 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>
```

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

For a better understanding of how XPath predicates are handled and how they are indexed, see [Chapter 13](#).

4.3 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 Database can be used to create event structures and rule classes that can process rules defined on spatial geometries. For this purpose, you can create a primitive event structure 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, you should associate the geometry metadata describing the dimension, lower and upper bounds, and tolerance in each dimension with each spatial geometry attribute. You can insert this metadata information 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 Developer's Guide*.

```
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 you use the event structure with spatial attribute 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>

```

You can form a composite event structure involving spatial attributes 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, you can use functions defined in the MDSYS.SDO_GEOM package to achieve this functionality. See *Oracle Spatial Developer's Guide* for additional information.

4.4 Rule Conditions for Text Events

You can configure the event structure associated with a rule class for one or more text attributes such that the corresponding rule conditions may include text predicates on these attributes. You specify the text predicates in the rule conditions using the Oracle Text CONTAINS operator. The CONTAINS operator refers to the text attribute defined in the event structure and applies a text query expression on the documents bound to this attribute.

You can define the text attributes in an event structure to be of CLOB or VARCHAR data type with associated text preferences. You can create such attributes using the DBMS_RLMGR.ADD_ELEMENTARY_ATTRIBUTE procedure. You specify the text preferences for an attribute using an instance of EXF\$TEXT type, which accepts the text preferences in string format, for example: (LEXER hotelreserv_lexer WORDLIST hotelreserv_wordlist). Use the preferences specified through this argument for parsing the document bound to the attribute and for indexing the text query expressions within the rule conditions. Alternately, you can assign an EXF\$TEXT instance with an empty preferences string to use default preferences.

For the travel services application considered in [Section 2.4](#), if each hotel reservation includes some addition information, you can model the AddHotel event structure as follows.

```

BEGIN
  DBMS_RLMGR.CREATE_EVENT_STRUCT (EVENT_STRUCT => 'AddFlight');
  DBMS_RLMGR.ADD_ELEMENTARY_ATTRIBUTE (
    EVENT_STRUCT => 'AddHotel',
    ATTR_NAME => 'CustId',
    ATTR_TYPE => 'NUMBER');
  DBMS_RLMGR.ADD_ELEMENTARY_ATTRIBUTE (
    EVENT_STRUCT => 'AddHotel',
    ATTR_NAME => 'Type',
    ATTR_TYPE => 'VARCHAR2(20)');
  . . .
  DBMS_RLMGR.ADD_ELEMENTARY_ATTRIBUTE (
    EVENT_STRUCT => 'AddHotel',
    ATTR_NAME => 'AddlInfo',
    ATTR_TYPE => 'CLOB',
    TEXT_PREF => EXF$TEXT('LEXER hotelreserv_lexer'));
END;

```

The rule conditions specified for the previous event structure can now include text predicates on the AddlInfo attribute shown as follows.

```
Type = 'Luxury' and CONTAINS (AddlInfo, 'Disney World') = 1
```

The event structure created with one or more text attributes could be part of composite event structure so that you can specify the text predicates on the individual primitive events as follows.

```
<condition>
  <and equal="Car.CustId, Hotel.CustId">
    <object name="Car"> carType = 'Luxury' </object>
    <object name="Hotel"> Type = 'Luxury' and
      CONTAINS (AddlInfo, 'Disney World') = 1 </object>
  </and>
</condition>
```

In the case of rules specified for composite events, the predicates involving text attributes are not allowed in the join clauses of the rule condition. The use of CONTAINS operator is only valid within the primitive event conditions. The text predicates in the rule conditions are processed using the CTXRULE index that is implicitly created for each text attribute. Unlike other forms of indexes, the CTXRULE index is not transactional in nature. That is, any modifications made to the rule conditions are not automatically reflected in the corresponding CTXRULE indexes. This could result in incorrect matching of events with rule conditions until you synchronize the text indexes with the updated rule conditions. You can synchronize all the text indexes associated with a rule class with the following command.

```
BEGIN
  DBMS_RLMGR.SYNC_TEXT_INDEXES (rule_class => 'CompTravelPromo');
END;
```

The user must have the EXECUTE privilege on the CTX_DDL package in order to run the previous command.

4.5 Disabling and Enabling Rules

Often there is a need to add rules to the rule class and keep them disabled. Such rules still belong to the rule class and they can be enabled at a later point in time. For this purpose, you can create a rule class table with DBMS_RLMGR.CREATE_RULE_CLASS procedure that implicitly has an `rlm$enabled` column in which to store the status of each rule. In the case of a rule class resulting from an upgrade of an Oracle 10g database, this column does not exist and thus the rules in such rule classes cannot be disabled.

The `rlm$enabled` column in the rule class table defaults to a value of Y to indicate that a rule by default is always enabled. Optionally, a value N can be assigned to this column during the insert of a new rule or update of an existing rule. A rule with N assigned to the `rlm$enabled` column is disabled and it cannot match any incoming events. Any modification to the `rlm$enabled` column (to enable or disable the rule) or a modification to the rule condition itself discards any intermediate state information associated with the rule. In effect, when an existing rule is enabled after being disabled, it is equivalent to a new rule with no prior state information.

4.6 Shareable Primitive Rule Conditions

The rules defined for composite events have conditions on individual primitive events and a join condition that relates these primitive events. In many rule-based applications, multiple rule conditions involving composite events use the same conditional expressions on the primitive events. For example, in the Travel Services

application described in [Section 2.4](#), a vacation in Orlando is defined as a round trip to Orlando with a minimum stay of 4 days (`ToCity = 'Orlando'` and `Return-Depart >= 4`). The same vacation in the Orlando scenario can be combined with a primitive event condition capturing a Luxury car rental and another condition capturing a car rental with a child seat option to form two different rule conditions. For such applications, Rules Manager has provisions to share parts of the rule condition across rules by using references to primitive rule condition repositories. Within a repository, each primitive rule condition has a unique identifier that can be referenced from one or more rules sharing the primitive condition. The ability to share primitive conditions simplifies the construction of the rule conditions for composite events and also allows managing them as one logical unit for any modifications made to the shared primitive rule condition.

The shareable rule conditions are associated with a primitive events structure and they do not belong to a particular rule class. So, a common list of rule conditions defined for a primitive event structure can be shared across multiple rules in a rule class as well as rules defined in multiple rule classes (configured with the same primitive event structure).

In addition to the basic steps for creating the event structure, the rule class, and the action callback procedure, a rules application with sharable primitive rule conditions has the following steps:

1. Create a primitive rule conditions repository for a given object type (to be configured as a primitive event structure) or an existing primitive event structure.

```
BEGIN
  dbms_rlmgr.create_conditions_table(
    cond_table    => 'FlightConditions',
    pevent_struct => 'AddFlight');
END;
```

The previous step creates a relational table to store the primitive rule conditions. This table is created with the user-specified name (`FlightConditions`) and it has a set of columns to store the unique identifier (primary key) for each rule condition (`rlm$condid`), the rule conditions (`rlm$condition`), and their descriptions in plain text (`rlm$conddesc`).

At the time of operation, if the primitive event structure specified for the primitive event argument is just an object type in the database and thus not associated with a rule class, the object type is converted into an event structure and it is later assigned as expression metadata to the `rlm$condition` column.

Note: There can only be at most one primitive rule conditions table associated with an event structure.

2. Insert a primitive rule condition into the table created in Step 1. The values stored in the condition column (`rlm$condition`) are automatically validated using the event structure associated with it.

```
INSERT INTO FlightConditions (rlm$condid, rlm$conddesc, rlm$condition)
VALUES ('OrlandoVacation', 'Vacation in Orlando',
       'ToCity = 'Orlando'' and Return-Depart >= 4');
```

3. Once you create one or more rule classes using the primitive event structure (`AddFlight`), the rules added to the rule classes can refer to the conditions in the corresponding primitive rule conditions table (`FlightConditions`). This is done

by assigning the primary key of a shared primitive rule condition to the ref attribute of the corresponding object element within the rule condition.

```
INSERT INTO CompTravelPromo
(rlm$ruleid, promoType, offeredBy, rlm$rulecond) VALUES
('PARKS_PROMO', 'TICKETS', 'THEME_PARKS_ASSOC',
 '<condition>
  <and join="Flt.CustId = Car.CustId">
    <object name="Flt" ref="OrlandoVacation"/>
    <object name="Car" ref="LuxuryCarRental"/>
  </and>
</condition>');
```

In the previous rule definition, the primitive rule condition references OrlandoVacation and LuxuryCarRental are resolved through the event structure names (AddFlight, AddRentalCar respectively) to the corresponding primitive rule condition tables and the primary keys within those tables. Multiple rules in the CompTravelPromo table can refer to the same primitive rule condition.

The primitive rule condition references within a rule condition for composite event are resolved at the time of rule definition. The runtime characteristics of such rules are similar to those of rules with in-place primitive rule conditions and the steps involved in evaluating the rules remain unchanged.

Note: When a primitive condition that is shared by one or more rules is updated, the changes are propagated to all the rules referring to it. Effectively, the rules themselves are considered updated and any intermediate state associated with these rules is discarded.

A primitive rule condition cannot be deleted when one or more rules in one or more rule classes refer to it. You cannot use a SQL DROP TABLE command to drop the primitive rule conditions table. You must use the DBMS_RLMGR.DROP_CONDITIONS_TABLE procedure instead. A primitive conditions table may not be truncated (with SQL TRUNCATE TABLE command) if a rule class is configured with the corresponding primitive event structure.

The list of rule conditions that have references to conditions defined in a conditions table can be obtained by joining the rule class table with the condition table using a join predicate with the DBMS_RLMGR.CONDITION_REF function. Given a rule condition and a primitive event name, this function returns the identifier (key) for the condition reference. This function returns NULL if the rule condition does not use a reference for that primitive event. The following query identifies all the rule conditions that refer to any shared conditions stored in the FlightConditions table.

```
select ctp.rlm$ruleid from CompTravelPromo ctp, FlightConditions fc
where dbms_rlmgr.condition_ref(ctp.rlm$rulecond, 'FLT') = fc.rlm$condid;
```

The previous query uses a functional index defined to retrieve rows from the rule class table based on the condition references they use. Note that if the rule class is configured for duplicate primitive events of the same type, the previous query should include multiple join predicates (combined with disjunctions). Also note that the primitive event name passed to the CONDITION_REF function must be case-sensitive for the previous query to benefit from the functional index. When a primitive event name requires quotes to preserve the case or special characters in the name, the quoted name must be passed to the CONDITION_REF function.

Note: When a rule class with one or more condition references is exported, the corresponding conditions tables are not automatically exported. The conditions table can be explicitly exported using the `tables` clause of the `EXPORT` command. The conditions tables are implicitly included in a schema export.

During an `IMPORT` operation, all the references in the rule conditions are resolved using the conditions table that existed prior to import or that are exported with the rule class. The import of a rule class fails with an `ORA-41704` error message when one of its rules refers to a missing conditions table or a condition. In such cases, the rules with invalid references are marked invalid by storing the value `F` under the corresponding `RLM$ENABLED` column.

4.7 Events Through Continuous Query Notification (Formerly Database Change Notification)

When the event structure used for a rule class is defined with one or more table alias attributes, the rule class can be configured to treat all `INSERT`, `UPDATE`, and `DELETE` operations on the underlying tables as the events for which the rules are evaluated (see [Section 3.7](#) and [Section 4.1](#)). You specify this using the `DMLEVENTS` property at the time of rule class creation. In this case, the row level triggers on the relational tables capture the events and invoke the `PROCESS_RULES` procedure from within the trigger body. As the rules are processed as part of DML operations, this configuration poses some restrictions on the types of rule actions possible (for example, rule actions cannot commit) and the `AUTOCOMMIT` policy for the rule class. Also, a long running transaction holding some locks on the matching rules in a rule class can often lead to deadlocks.

Identifying the events after committing the DML operations is often desirable to avoid the side effects of `DMLEVENTS` configuration. Rules Manager provides an option to process the rules for the net data changes within a transaction after the end of the transaction. In this case, Rules Manager makes use of the continuous query notification (formerly database change notification) feature (see *Oracle Database Advanced Application Developer's Guide*) to receive notifications of net data changes (within a transaction) after the end of each transaction. These notifications are used to capture the modified rows or the event data and match them with the rules in the rule class.

A rule class using an event structure with one or more table alias attributes can be configured for continuous query notification (CNF) events with the `CNFEVENTS` property at the time of rule class creation. The rule class can be configured to treat just `INSERT` notifications or all `INSERT`, `UPDATE`, and `DELETE` notifications as events with `CNFEVENTS="I"` or `CNFEVENTS="IUD"` specifications respectively. The configuration `CNFEVENTS="I"` is ideal for append-only databases. The user creating a rule class with the `CNFEVENTS` property should be granted the `CHANGE NOTIFICATION` privilege and the execute privilege on the `DEMS_CHANGE_NOTIFICATION` package. A rule class cannot be configured for both DML events and CNF events. Using a combination of table alias and embedded primitive event types in the composite event structure, a rule class can be configured to obtain a subset of the events automatically through continuous query notification while others are explicitly added by the application.

Unlike in the case of `DMLEVENTS`, where the rules are processed synchronously with a DML operation, a `CNFEVENTS` configuration processes the rules asynchronously after

the commit of a transaction. Since the order in which the continuous query notifications for a set of transactions are processed is not guaranteed, rules that are sensitive to the order of events may yield inconsistent results. Also, the version of the row picked up at the time of rule evaluation could be different from the version of the row at the end of transaction (with potential race conditions). By picking the latest version of the row for rule evaluation, it is ensured that the rule class with CNFEVENTS="IUD" configuration has the correct persistent state information after all the events (notifications) are processed.

4.8 Collection Events

A class of rule-based applications requires support for conditional expressions on collections of events as opposed to individual events. The rule conditions in such applications compute aggregate values over a finite but potentially large number of primitive events of the same type and specify predicates on the resulting aggregates. For this purpose, primitive events of a specific type are grouped based on certain event attributes and aggregate operators such as SUM, AVG, MIN, MAX, and COUNT on the other event attributes are used to apply predicates. For example, a rule condition may test the sum of amounts transferred from a particular account. In this scenario, if each bank transaction is an individual event, these events are grouped based on the account identifier and the predicate is specified on the sum of the amounts (SUM(amount) > 10000).

Rules Manager supports aggregate operators in rule conditions with the use of collection events. A collection event consists of one or more primitive events of a specific type grouped by one or more event attributes and optionally satisfying user-specified conditions described with the having clause and/or a text node within the collection element. For example, a collection event could represent a set of BankTransaction events that share a common account identifier attribute and for a withdrawal type. The items needed to define a collection event include at least one primitive event structure, a composite rule class enabled for collection events, a composite event, and a named collection specified by a <collection> element in a rule condition as follows:

```
<condition>
  <collection name="bank" groupby = "subjectId"
    having = "SUM(amount) > 10000">
    tranType = 'Withdrawal'
  </collection>
</condition>
```

The name attribute of the <collection> element restricts the type of primitive events considered for the collection and the value specified for the groupby attribute combined with predicates on the other event attributes define the properties that are common across all the primitive events in the collection. The having attribute specifies the aggregate predicates that should be evaluated for the events in the collection. A rule condition involving a <collection> element can be viewed as operating as a standard SQL query with WHERE, GROUP BY and HAVING clauses. For example, the preceding rule condition can be mapped to the following SQL query on the conceptual Transactions table.

```
SELECT DISTINCT subjectId FROM Transaction
WHERE tranType = 'Withdrawal'
GROUP BY subjectId
HAVING SUM(amount) > 10000
```

In the preceding rule condition, each bank transaction primitive event is tested for the `Withdrawal` transaction type and the matching primitive events are grouped based on their `subjectId` attribute resulting in one collection event for each distinct `subjectId`.

4.8.1 Setting a Fixed Window Size or Length to Restrict the Number of Primitive Events

The primitive events participating in a particular collection may be further restricted using moving window semantics. Rules Manager supports two types of moving windows with collection events:

- Fixed window length: The window length for a collection is specified as n units of time (fraction of a day). A collection with a fixed window length only consists of the primitive events that are generated within the last n units of time. Any primitive event generated prior to this time are not considered for evaluating the aggregate conditions. With a `windowlen="1"` specification in the following example, the SUM is computed only for the bank transactions that are generated within 24 hours of the last event in the collection. This rule identifies the situation where over \$10,000 is withdrawn from an account within the past 24 hours.

```
<condition>
  <collection name="bank" groupby = "subjectId"
    having = "SUM(amount) > 10000"
    windowlen = "1">
    tranType = 'Withdrawal'
  </collection>
</condition>
```

- Fixed window size: The window size for a collection is specified as n number of primitive events. A collection with a fixed window size specification only consists of the most recent n primitive events that belong in that collection. With a `windowsize="10"` specification in the place of the `windowlen` specification in the preceding example would compute the SUM for the last 10 withdrawal transactions for each account.

4.8.2 Using Collection Primitive Events to Form Composite Events

With some restrictions, a rule condition can combine a collection event with other collection events or primitive events to form composite events involving collections (see [Section 5.6](#)). You can use the basic rule condition syntax for composite events to relate one collection with other events within a composite event.

Unlike in the case of rule conditions involving set semantics (see [Section 5.4](#)), the number of primitive events participating in the rule conditions through collections is not restricted by the event structure definition. A subset of the primitive event types within a composite event are configured for collections and each rule condition in the corresponding rule class can consider varying number of primitive events using varying window specifications. Consider a Law enforcement application that relates `BankTransaction`, `Transportation`, and `FieldReport` events to raise some security alerts (see [Section 10.1](#)). In this application, if there is a need to specify aggregate predicates on `BankTransaction` events, the rule class created with the `LawEnforcement` composite event structure should be configured as follows.

```
BEGIN
  DBMS_RLMGR.CREATE_RULE_CLASS(
    rule_class => 'LawEnforcementRC',
    event_struct => 'LawEnforcement',
    action_cbk => 'LawEnforcementCBK',
```

```

actprf_spec => 'actionType VARCHAR2(40), actionParam VARCHAR2(100)',
rslt_viewnm => 'MatchedCriteria',
rlcls_prop =>
  '<composite
    equal="bank.subjectId, transport.subjectId, fldrpt.subjectId"
    ordering="rlm$rule.rlm$ruleid, bank.subjectId, transport.subjectId">
    <collection type = "BankTransaction"
      groupby = "subjectId, tranType"/>
  </composite>');
END;
/

```

For each primitive event type participating in collections, the rule class properties should include a `<collection>` element with the name of the type and a `groupby` specification. The `groupby` specification lists all possible `groupby` attributes expected with this primitive event. The rule class created with the preceding command can include rule conditions that group bank transactions just on the `subjectId` or `tranType` attributes or a combination of these two attributes, (`subjectId, tranType`). Note that grouping of events could also be based on some expression involving the attributes in the event structure. For example, if some rules need to group the events based on the region from which the events originated, a user defined function, which maps a country to a broader region can be included in the `groupby` specification. (Example: `groupby = "subjectId, tranType, regionOf(fundFrom) "`).

The use of a `<collection>` element in the place of an `<object>` element in rule class properties enables the specific event type for collections. The `<collection>` element can also include the attributes that are allowed with the `<object>` elements such as `consumption` and `duration`. A rule class configured for collection events internally uses some additional database objects to maintain the incremental state information pertaining to collections. Such rule classes can store and manage rule conditions involving individual events as well as those involving collection events.

When a rule class is configured for collections involving specific primitive event type, the corresponding rules can create collection events out of individual primitive event instances using the `<collection>` element within the rule condition syntax for composite events. The rule condition syntax for collection events has provisions for grouping a set of primitive events to form collection events and for testing aggregate predicates on these events. The attributes over which the primitive events are grouped is a subset of the attributes listed in the rule class properties for the given event type. Each rule in the rule class may use a different `groupby` specification for the collection event. The aggregate predicates for the collection events are specified using the SQL operators `COUNT` for counting the number of events, `AVG` for computing the average value for an attribute, `SUM` for computing the sum of certain attribute, `MIN` for computing the minimum values for an attribute, and `MAX` for computing the maximum values for an attribute. The aggregate predicates expressed in SQL-HAVING clause syntax are assigned to the `having` attribute of the `<collection>` element.

```

<condition>
  <collection name="bank" groupby = "subjectId"
    having = "SUM(amount) > 10000 or COUNT(*) > 10"
    windowlen = "1">
    tranType = 'Withdrawal'
  </collection>
</condition>

```

Unlike the primitive events that are shared across multiple rules that match the event, a collection event is specific to the rule matching it. Based on the number of unique combinations of the attributes the primitive events are grouped on, one rule condition may have multiple collection events associated with it. These collection events are maintained as some primitive events are added to the collection or (dropped for the collection owing to window specification). Each collection event and the corresponding aggregate values are computed incrementally and the resulting state is stored persistently in the database. Note that the aggregate values are computed only upon the arrival of a new event and a primitive event dropping out of a window due to elapsed time does not force the computation.

When a rule condition involving collections evaluates to true, the corresponding action can be executed using the action callback mechanism or the results views as discussed in [Section 2.4](#) and [Section 2.6](#). For the action execution, an instance of the collection event is passed into the callback procedure. The collection event is of the same type as the primitive events of which it consists. However, in the collection event, only the attributes on which the events are grouped (native attributes from the collections's `GROUP BY` clause) are initialized, while the rest are each set to `NULL`. For example, a collection event matching the previous rule condition will be an instance of `BankTransaction` type that only has the `subjectId` (known to be common across all primitive events in the collection) initialized. In order to provide access to the aggregate values computed for the collection event, the action callback procedure (and the results view) is created with an additional argument for passing in the collection event identifier (`ROWID` type). This event identifier is passed into the `DBMS_RLMGR.GET_AGGREGATE_VALUE` call to fetch the computed values for an aggregate function such as `SUM(amount)`.

```
CREATE OR REPLACE PROCEDURE LawEnforcementCBK (
    bank      BankTransaction,
    bank_evtid ROWID,    -- event identifier for the "bank" collection events --
    transport Transportation,
    fldrpt    FieldReport,
    rlm$rule  LawEnforcementRC%ROWTYPE) IS
BEGIN
    ..
    dbms_rlmgr.get_aggregate_value(rule_class => 'LawEnforcementRC',
                                   event_ident => bank_evtid,
                                   aggr_func  => 'SUM(amount)');
END;
```

Note that the `DBMS_RLMGR.GET_AGGREGATE_VALUE` call returns a non-null value only if the signature of the aggregate function passed in matches one of the aggregate operators computed for that collection.

4.8.3 Collection Events Have NULL Values Stored in its `rlm$CrtTime` (Timestamp) Attribute

Since the collection event is created over a period of time, there is no precise timestamp associated with such an event. A null value is stored in its `rlm$CrtTime` (timestamp) attribute and thus operations involving this attribute (such as `SEQUENCE`) are invalid. For the same reason, a duration policy that is specified as elapsed time has no impact on collection events. Note that when you configure the rule class with `SESSION` or `TRANSACTION` event duration policies, the collection events are also expired at the end of `SESSION` or `TRANSACTION` respectively. Since a collection event is private to a rule, consuming such an event (using an `EXCLUSIVE` or `RULE` consumption policy) will reset the collection (empty the collection) and does not impact the number of rules that are executed. Recall that when multiple rules evaluate

to true with a primitive event (no use of collections), consuming the event by one or more of these matching rules will automatically stop the action execution for the other rules.

When the rule conditions with collection constructs coexist with the rule conditions with individual primitive events (no collections), the consumption of the primitive events owing to the `EXCLUSIVE` or `RULE` consumption policy does not drop the primitive event from the existing collection events. Similarly, when the primitive events expire due to an elapsed time duration policy, the collection events depending on this event are not impacted.

A rule class can be configured for multiple primitive event types that are collections. However, in this release, the primitive events modeled as table aliases to some relational table cannot be configured as collections. Also, the composite event structure used for the rule class may not have multiple primitive events of the same type.

4.9 Performance Tuning

Tuning a rule class for optimal performance falls into three broad areas:

- Choosing the appropriate `EQUAL` property

In the case of a rule class for composite events, identifying the most-common equality join predicates in all the rule conditions and specifying them using the `EQUAL` rule class property is important for performance. Different forms of `EQUAL` property specifications and their syntax is discussed in [Section 3.4](#).
- Tuning Expression Filter indexes

The indexes used to identify the candidate rules that match a given event can be tuned for better performance.
- Creating Rule Class Interface package

A rule class specific interface package helps reduce the overhead involved with the generic interfaces.

Tuning Expression Filter Indexes

A rule class created for simple or composite events implicitly creates one or more Expression Filter indexes to process the rule conditions for incoming events. These default indexes, created at the time of rule class creation, assume that predicates with all scalar attributes in the event structures are equally likely in the rule conditions and this assumption may not hold true for most applications. The performance of a rules application can be improved by tuning the indexes to the specific workload. This is possible either with the domain knowledge or by analyzing a representative workload from the rules application.

When using the domain knowledge to tune the Expression Filter indexes created for a rule class, treat the primitive event structures (or a simple event structure) as the Expression Filter attribute sets (see [Section 11.2](#)).

The default index parameters associated with an attribute set can be obtained by querying the `USER_EXPFIL_DEF_INDEX_PARAMS` view and they can be changed using the `DBMS_EXPFIL.DEFAULT_INDEX_PARAMETERS` call (see [Section 12.6](#)). These default index parameters are used any time a corresponding Expression Filter index is created. The default indexes created for a rule class can be dropped using the `DBMS_RLMGR.DROP_EXPFIL_INDEXES` call and they can be recreated to use the user assigned index parameters using the `DBMS_RLMGR.CREATE_EXPFIL_INDEXES` call.

When using a workload to tune Expression Filter indexes created for a rule class, the most common predicate constructs in the rule conditions can be identified by collecting statistics on a representative set of rules already defined in the rule class. The Expression Filter indexes are created from these statistics by setting the `coll_stats` argument of the `DBMS_RLMGR.CREATE_EXPFIL_INDEXES` call to `YES`.

When the event structure used for a rule class has one or more XMLType attributes, the default Expression Filter indexes and the indexes created from statistics are not optimized for the XPath predicates on these attributes. Hence, the XPath index parameters should be explicitly assigned to the event structure using the `DBMS_RLMGR.DEFAULT_XPINDEX_PARAMETERS` call as discussed in [Section 13.2.4](#).

Create Rule Class Interface Package

A way to improve the runtime performance of a rules application is to create a rule class interface package that is specific to the rule class. This avoids the overhead involved in using the generic `DBMS_RLMGR` package. The `DBMS_RLMGR` procedures used for the run time operations such as processing the rules for some events, consuming the events and resetting the session make use of the rule class name passed in as one of the arguments and map them to the corresponding operations on the rule class. This step can be avoided by creating a rule class interface package that is used to directly operate on the rule class. The rule class interface package is most effective when the rules in the rule class are selective or they often do not match any incoming events. This package is created using the `DBMS_RLMGR.CREATE_INTERFACE` call and this has a set of procedures to perform all runtime operations on the rule class. See the `DBMS_RLMGR.CREATE_INTERFACE` call in *Oracle Database PL/SQL Packages and Types Reference* for additional information.

4.10 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 are 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.

4.11 Resetting Events for Development Environments

When developing rule applications using Rules Manager, rules defined in the rule class can be tested using some hypothetical events. Because these events could match some rules in the rule class partially, they could contribute to the incremental states stored in the database. So before you deploy your application in a production environment, you must purge this partial state information so as not to cause any unexpected rule actions. Use the `DBMS_RLMGR.PURGE_EVENTS` procedure call to purge any state information and the events from the database prior to deploying the application in a production environment.

Rule Conditions

Rules Manager rule conditions use the SQL `WHERE` clause format and the attributes of the event structure to define the rule condition. For the travel services example, you express the rule condition using the attributes: `Airline`, `ToCity`, `Return`, and `Depart`. Rule conditions you define on primitive event structures correspond directly to the SQL `WHERE` clause format:

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

Note that you embed all rule conditions within XML `<condition>` tags. Rules Manager provides additional XML tags to support incremental evaluation of rule conditions for composite events (which are composed of two or more primitive events).

You may recall that evaluating a condition based on a primitive event is atomic, implying that the values for all attributes of that event structure are available atomically. Thus, when you define a rule condition on a primitive event, it evaluates to true or false instantaneously. In contrast, when you define a rule condition on a composite event, it may have intermediate states, depending on the subset of the primitive events that are available. For example, when you define a rule on a composite event constructed from three primitive events, you can define it to fire if any two of the three primitive event conditions are true.

5.1 Support for Incremental Evaluation of Rules

To support rule conditions on composite events, you can use additional XML tags within the `<condition>` tags. These tags extend the basic `WHERE` clause functionality, supporting joins between composite events, and incremental evaluation of the primitive event instances that comprise the composite event structure.

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, you can represent the previous rule condition 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. You can insert the rule condition in this format into the `rlm$rulecond` column of the corresponding rule class table. Rules Manager provides XML tags to support more complex rule constructs. Figure 5–1 and Table 5–1 summarize these tags and Section 5.2 through Section 5.5 describe these tags in more detail.

The previous example illustrates the combination of **and** with a **join**, which evaluates to true when all the primitive event conditions evaluated to true with the corresponding events, which also satisfy the rule's join condition. Other constructs, such as **any**, enable complex conditions to be specified that evaluate to true if a subset of the primitive event conditions are true.

The most common join predicate used to form composite events is an equality predicate, as is the case with SQL queries that join multiple tables. Usually, one or more attributes from each primitive event are compared with one or more attributes from the other events for equality. Rules Manager uses a convenient syntax to specify the equality join predicates in the rule conditions and also provides a mechanism to enforce this join predicate for all the rules in a rule class (Section 3.5).

The following are examples of commonly used rule constructs defined on composite events.

- Count based subsets of primitive events. Specify by using the **any** operator.
 - The operator **any** evaluates to true if any of the primitive event conditions evaluate to true.
 - The operator **any 2** evaluates to true if any 2 or more of the primitive event conditions evaluate to true.

In general, the **any** operator is parameterized with a **count** argument, which evaluates to true if **any count** of the primitive event conditions evaluate to true.
- Sequenced subsets of primitive events. Specify these in a number of ways:
 - Use a **Join** with a time constraint. Use the time constraint to impose a partial order over the event instances.
 - Use combinations of **and** with a **sequence** tag.

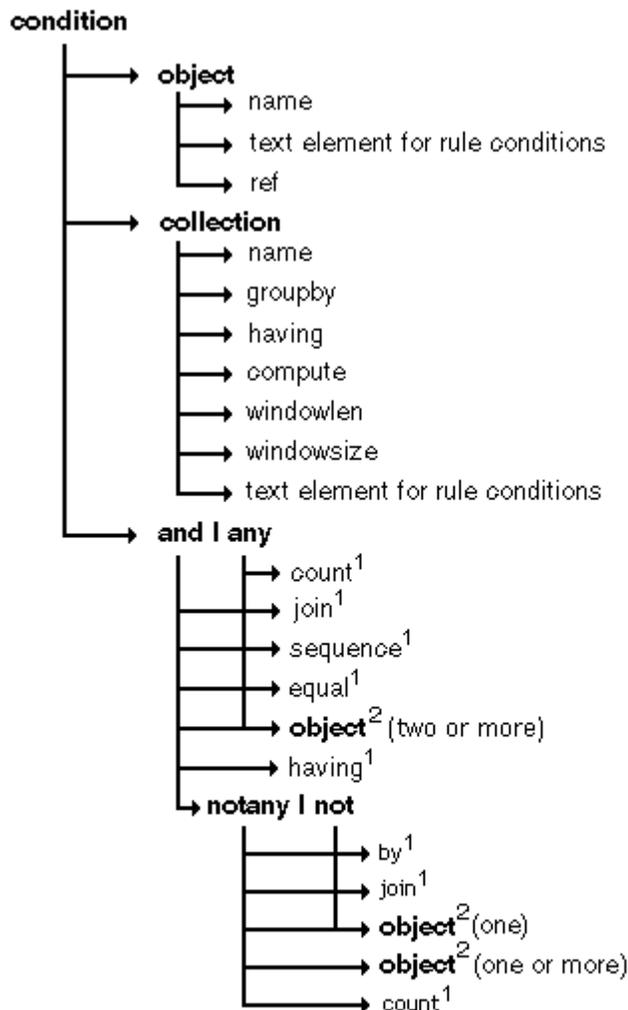
The sequence tag requires a specific ordering of primitive events.

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 you must specify using an entity reference (`<`) or use 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.

- Use combinations of **any count** with a **sequence** tag.
- Detecting if one primitive event does not occur within a certain time interval of another primitive event. Specify this using the **by** option of the **not** or **notany** tag:
 - Use the **not by** tag with a timestamp parameter to detect the non-occurrence of a primitive event within a specific time interval.
 - Use the **notany by** tag in a similar fashion.

Figure 5–1 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 the Rule Condition section in Appendix F. Table 5–1 shows a relational view of the same supported XML tag extensions showing the XPath and some notes about the elements and attributes.

Figure 5–1 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 5–1 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	Two or more objects	
		notany	notany/object	One object	
collection	Element	condition	condition/collection	One	Collection event
		and	and/collection	Two or more in combination with object	
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
		collection	collection/@name		Collection name

Table 5–1 (Cont.) Relational View of the XML Tag Extensions

XML Tag	Type	Parent	XPath	Number of Occurrences Allowed within Its Parent	Notes
ref	Attribute	object	object/@ref	One	Reference to a shared condition
groupby	Attribute	collection	collection/@groupby	One	Group by specification for the collection
having	Attribute	collection	collection/@having	One	Having clause for the collection
	Attribute	and	and/@having	One	Having clause joining multiple collections
compute	Attribute	collection	collection/@compute	One	Additional aggregate function to compute for the collection
windowlen	Attribute	collection	collection/@windowlen	One	Moving window spec for collection
windowsize	Attribute	collection	collection/@windowsize	One	Moving window spec for collection

5.2 Rule Conditions with Sequencing

The rules you define 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 uses the implicit timestamp attribute (`rlm$CrtTime`) that is included in each primitive event participating in a composite event.

Rules Manager uses the event creation times in the primitive events to enforce and detect sequencing in rule applications. For example, consider the rule in the travel services application. You can specify an additional predicate to offer the promotion only if the `AddRentalCar` event is generated after the `AddFlight` event. You can extend the rule condition 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. You can replace the sequence attribute 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>
```

You can use sequencing 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). You can also use the `rlm$CrtTime` attribute in the primitive event type 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. The boldfaced text of the following example indicates where the value 1 means one day. See *Oracle Database Advanced Application Developer's Guide* for more information about performing date or 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, Rules Manager treats the `rlm$CrtTime` attribute as any other attribute in the event structure. However, when you do not specify a value 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 rule class.

5.3 Rule Conditions with Negation

Typically, you use rules with negation in their conditions 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, you define the rule 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, you can express the rule condition for the order tracking rule as shown in the following example. The boldfaced text, "`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.
- Use the `<notany>` element in place of the `<not>` element to support a notion of disjunction within the negation rule.
- At the time of activation, Rules Manager executes the `by` attribute of the `<not>` element to compute the deadline for the primitive events in the `<not>` element. You can express the value for the `by` attribute 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 5.2](#)), 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, you can also express the rule condition in the preceding example 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 you expect the shipment 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 `AUTO COMMIT` property turned off (see [Section 3.6](#)). This also includes the rule classes configured for `DMLEVENTS` (see [Section 3.7](#)).

You can use rules involving negation constructs 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. You can use a `<not>` element without a hard deadline (no `by` attribute) to define such rules.

Consider a composite event with three primitive events: `PlaceOrder`, `PaymentReceived`, and `ShipOrder`. You can use a rule 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, you can represent the composite event structure and the rule condition 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). You can use the `sequence="yes"` (Section 5.2) property, such as shown in the following example, 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, you determine the deadline for the `PaymentReceived` event by the occurrence of the `ShipOrder` event, which follows the corresponding `PlaceOrder` event. In effect, if the `ShipOrder` event is detected before the `PaymentReceived` event for a particular order, the rule application executes the action associated with the preceding rule condition.

You can often use the negation construct 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`. You can express such rule conditions 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, you can use primitive events 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), you can use the `join` attribute for the `<not>` element. 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).

5.4 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. You can define rules 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, you model the composite event structure 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, you can configure the events for SESSION duration (see [Section 3.3](#)) such that only the primitive events generated within a user session are considered for rule matches. Using the composite event rule condition syntax, you can express the preceding condition 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 you use the element names `Item1`, `Item2`, and `Item3` 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>
```

The maximum number of primitive events allowed in a composite event limits the total number of primitive events you can consider in a rule condition with set semantics. Also, following standard SQL semantics, you cannot use aggregate operators in the join conditions to relate multiple events. For rule-based applications involving aggregate operators over a finite, but potentially large number of primitive events, you should configure the rule class for collection events and the rule conditions should specify predicates on these collections.

5.5 Rule Conditions with Any n Semantics

The examples discussed so far use rules that match all the primitive events specified in a rule condition. You achieve this 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 you define for the composite event may need to match only one of the three primitive events. For this example, you represent the composite event structure and the rule condition 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, use the `count` attribute of the `<any>` element, 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 5.4](#) can be extended to suggest the tripod to the customer if the shopping cart has any two of the three items specified. You can represent the condition for this rule 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. You can represent this new rule condition 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, you can extend the preceding rule 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), you can pose this 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, you cannot use similar join predicates 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, you can represent the preceding join condition 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 supports this abbreviated syntax 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 you use both `equal` and `join` attributes 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 3.4](#).

You can use the `sequence` attribute ([Section 5.2](#)) in an <any> element to ensure that the matching primitive events happen in the specified order for the rule condition to evaluate to true, shown as follows.

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

```

    <object name="Item3"> Accessory = 'IR Light' </object>
  </any>
</condition>

```

5.6 Rule Conditions with Collection Events

A collection is an event instance formed by grouping a set of primitive events based on some common properties. The common properties shared by the primitive events could be the equality of certain attributes (for example, all the events in a collection have the same identifier) and they can also include some predicates on the content of the primitive events (for example, all primitive event in the collection satisfy the predicate `tranType = 'Withdrawal'`).

Consider a rule class configured for three types of primitive events, BankTransaction, Transportation and FieldReport out of which the BankTransaction events are enabled for collection (see [Section 4.8](#)). The rule class can now include rule conditions that test some aggregate predicates on bank transaction events - specifically, bank transaction collection events. For example, the following rule condition makes use of the collection element within the rule condition syntax to test some aggregate predicates.

```

<condition>
  <collection name="bank" groupby="subjectId"
    having="SUM(amount) > 10000">
    tranType = "Withdrawal" and amount > 1000
  </collection>
</condition>

```

With the preceding rule condition, you can consider for collection all the primitive events that match the criteria specified as the collection element's text value (`tranType = "Withdrawal" and amount > 1000`). The rule condition uses the value specified for the `groupby` attribute of the `<collection>` element to create multiple collection events, one for each unique subject identifier (`subjectId`). The collection event maintains the summaries of the primitive events that formed it and loses the identities of the individual events. It uses these summaries to compute the aggregate values necessary to evaluate the predicates specified in the `having` clause (value assigned to the `having` attribute) of the `<collection>` element. As new primitive events occur, it computes necessary aggregate values incrementally and the predicates in the `having` clause are evaluated. When the condition specified in the `having` clause evaluates to true, it considers the rule condition as true and the action associated with the rule is executed. By default, Rules Manager executes the action (or invokes the action callback procedure) synchronously with the last event (time ordered) in the collection. For each new event in the group that keeps the condition in the `having` clause true, Rules Manager executes the rule action once. Alternately, you can reset the collection after executing the action by using `EXCLUSIVE` or `RULE` consumption polices (see [Section 4.8](#)).

The conditional expression you specify for the `having` clause of the rule condition is in SQL `HAVING` clause format with one or more predicates joined by conjunctions and disjunctions. You form the predicates in the `having` clause using one of the five aggregate operators, `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX`, each operating on one of the attributes from the corresponding event structure. Some sample forms of SQL `HAVING` clause specification are as follows.

```

SUM(amount) > 10000 and AVG(amount) >= 1000
SUM(amount) > 10000 and (AVG(amount) >= 1000 or COUNT(*) < 10)
MAX(amount) > 5000 or MIN(amount) > 1000

```

5.6.1 Setting a Fixed Window Size or Length to Restrict the Number of Primitive Events

With the previous rule condition syntax, the number of primitive events participating in the collection is monotonically increasing until the collection is reset (by consuming the collection event). Optionally, you can restrict the number of primitive events participating in a collection by using one of the two moving window semantics:

- Fixed window size
- Fixed window length

The fixed window size specification for collection events enforces a maximum limit on the number of primitive events in a collection by dropping the oldest event with the addition of a new event. For example, you can enhance the previous rule condition to keep summaries for only the last 100 events (time ordered from the last event occurring) in each collection shown as follows.

```
<condition>
  <collection name="bank" groupby="subjectId"
    having="SUM(amount) > 10000"
    windowsize="100">
    tranType = "Withdrawal" and amount > 1000
  </collection>
</condition>
```

Alternately, a rule condition can restrict the primitive events in a collection by using a fixed time window, which ends with the last primitive event added to the collection. For example, you can write the previous rule condition to consider only the events occurring in the past 24 hours, shown as follows. You express the window length specification as a fraction of a day.

```
<condition>
  <collection name="bank" groupby="subjectId"
    having="SUM(amount) > 10000"
    windowlen="1">
    tranType = "Withdrawal" and amount > 1000
  </collection>
</condition>
```

When a rule condition with collection constructs evaluates to true, a reference to the collection event instance representing a set of primitive events is returned for action execution (in the action callback procedure or the results view). The collection event is of the same type (`BankTransaction` in previous examples) as the primitive events of which it consists. However, in the collection event, Rules Manager only initializes the attributes on which the primitive events are grouped (native attributes from the collection's `group by` clause), while the rest are each set to `NULL`. For example, the collection event created for the previous rule condition is a `BankTransaction` event with just the `subjectId` attribute initialized. The rest of the attributes, potentially being different for each primitive event in the collection, are each set to `NULL`.

For each collection event that is made available to the action logic, you can obtain the aggregate values computed for the collection by using the collection event identifier and the `DBMS_RLMGR.GET_AGGREGATE_VALUE` call (see [Section 4.8](#)). If the action logic relies on more aggregate values than those that are computed for applying predicates, you can specify them using the `compute` attribute of the `<collection>` element. For example, the following rule condition computes the minimum amount value within each collection (in addition to the sum of amounts), which can be fetched into the application at the time of action execution. Each collection event in a rule condition can maintain a total of 5 aggregate values.

```
<condition>
```

```

<collection name="bank" groupby="subjectId"
  having="SUM(amount) > 10000"
  compute="MIN(amount) "
  windowlen="1">
  tranType = "Withdrawal" and amount > 1000
</collection>
</condition>

```

You can use the rule condition syntax for composite events to relate a collection event with another collection event or a primitive event. For example, you can relate a collection event representing a set of bank transactions to a transportation event to form a rule such as "if a subject withdraws over \$10,000 in a day and he rents a truck one-way into a restricted area, add him to the NYPD watch list."

```

ON
  BankTransaction(subjectId, amount, tranType, ..) bank,
  Transport(subjectId, vesselType, locFrom, locTo, ..) transport
IF
  <condition>
    <and equal="transport.subjectId, bank.subjectId">
      <collection name="bank" groupby="subjectId"
        having="SUM(amount) > 10000"
        windowlen="1">
        tranType = "Withdrawal"
      </collection>
      <object name="transport">
        vesselType = 'TRUCK' and locTo != locFrom and IsRestrictedArea(locTo) = 1
      </object>
    </and>
  </condition>
THEN
  PerformAction('ADD2WATCHLIST', 'NYPD', subjectId)

```

While relating a collection event with other collections or individual (non-collection) events or both, the attributes listed in its `groupby` clause can be used to form join predicates (`join` and `equal` clauses) across events. For example, a collection event formed with the previous rule has its `subjectId` attribute initialized to the subject identifier that is common across all the primitive events in the collection (owing to the collection's `group by` clause) and this attribute can be used to join the collection event (bank) with the other events.

The rule application executes the action associated with the previous rule when a bank transaction collection event and a transportation event, both meeting their corresponding criteria, match on their `subjectId` attribute. Depending on the order in which the primitive events occur, it executes the action for this rule synchronously either with the following:

- The last bank transaction event in the collection that satisfies the `having` clause when there is already a transportation event that matches the other criteria, or
- The transportation event that satisfies its criteria and bears the same subject identifier as a bank transaction collection event that satisfied its `having` clause.

The `having` attribute in the `<collection>` element can only include predicates involving a single collection. If the conditional expression in the `having` clause should relate multiple collections or one collection with other events, you should use the `having` attribute of the `<and>` element. For example, in the following rule condition managing items in a crate (both considered to be radio-frequency ID (RFID) read events), Rules Manager groups the item primitive events based on their `crateid`

attribute and compares the aggregate value computed with the `capacity` attribute of the crate.

5.6.2 More About Using the Having Clause

```
<condition>
  <and equal="crate.id, item.crateid"
    having="COUNT(item.*) > crate.capacity*0.8">
    <object name="crate"/>
    <collection name="item" groupby="crateid"/>
  </and>
</condition>
```

The `having` clause specified with the `<and>` element can refer to any collection or object you define within the `<and>` element using extended names (such as `item.*` and `crate.capacity`). In such conditions, the `having` clause acts as the join condition between collections and other primitive events. This combines the aggregate predicate acting as join conditions with the aggregate predicates on specific collections. However, Rules Manager only optimizes for faster evaluation the aggregate predicates you specify with the `having` attribute of a `<collection>` element. For example, you can extend the previous rule condition to include a predicate on the minimum number of items in a crate, shown as follows, and with this rule, you can consider a collection event for further evaluation only when it has over 50 individual events.

```
<condition>
  <and equal="crate.id, item.crateid"
    having="count(item.*) > crate.capacity*0.8">
    <object name="crate"/>
    <collection name="item" groupby="crateid"
      having="COUNT(*) > 50"/>
  </and>
</condition>
```

You can also use the ability to specify aggregate predicates in the `having` clause of the `<and>` element to relate multiple collections. For example, you can compare two collections of primitive events, one representing the withdrawals and the other representing the deposits, to check the trend on a bank account over a period of time.

```
ON
  Deposits (subjectId, amount, ..) dep,
  Withdrawals (subjectId, amount, ..) wdr
IF
  <condition>
    <and equal ="dep.subjectId, wdr.subjectId"
      having = "SUM(wdr.amount) > SUM(dep.amount)">
    <collection name="wdr" groupby="subjectId" windowlen="30"/>
    <collection name="dep" groupby="subjectId" windowlen="30"/>
  </and>
  </condition>
THEN
  Alert (dep.accountId, 'Negative Trend');
```

In current release, you cannot combine collection constructs in a rule condition with rule conditions with Negation (see [Section 5.3](#)) or Any constructs (see [Section 5.5](#)).

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. You can query the results view to determine the rules that match an event and their corresponding actions can then be executed.

6.1 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 uses 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$ruledesc  VARCHAR2(1000),
    rlm$enabled   CHAR(1) DEFAULT 'Y');
);
```

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),
    rlm$enabled   CHAR(1) DEFAULT 'Y');
```

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;
```

Use the results from this query 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 3.2](#) 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, you can use the reset session procedure (`dbms_rlmgr.reset_session()`) 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. You can mark an event 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, Rules Manager automatically handles the consumption of the exclusive events. However, when you use external action execution, 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, you should execute the action for a particular rule 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;
```

Obtain the event identifier from the value listed in the `rlm$eventid` column of the `MatchingPromos` results view. If the consumption policy (see [Section 3.2](#)) for all events is shared, then the `CONSUME_EVENT` call always returns 1 and the event is still available. Note that only the events you configure for exclusive

consumption are consumed and the corresponding rows from the results view are deleted.

6.2 Modes of Operation

Designing a rule-based application using Rules Manager involves 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, you no longer need to deal with the Rules Manager APIs. You add new rules using the SQL `INSERT` statement against the rule class table and the run-time calls that are embedded in larger applications 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 6.2.2](#)).

6.2.1 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.

6.2.2 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 you can use single tier mode with a few modifications (thereby reducing the number of steps involved to two). You can implement the action callback procedure in the single-tier mode 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. This configuration requires a minimum of two database calls (`PROCESS_RULES` call and `DEQUEUE` call).

Rules Manager, as a database feature, works 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 rules 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. You can also use this view 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).

6.2.2.1 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, it returns this SOAP message 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 contains one predefined type and a public synonym for this type. [Table 7-1](#) describes the Rules Manager object type.

Tip: See the chapter on Rules Manager Types in *Oracle Database PL/SQL Packages and Types Reference* for all reference information concerning Rules Manager object types. Use the link in the Object Type Name column in [Table 7-1](#) to see the object type.

Table 7-1 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. To preserve the case, enclose the values with double quotation marks.

Tip: See the chapter on the DBMS_RLMGR package in *Oracle Database PL/SQL Packages and Types Reference* for all reference information concerning Rules Manager package procedures. Use the links in the Procedures column in [Table 8-1](#) to see each procedure.

Table 8-1 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 and to the Expression Filter attribute set
ADD_RULE procedure	Adds a rule to the rule class
CONDITION_REF function	Retrieves the primitive rule condition reference from a rule condition for composite events
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_CONDITIONS_TABLE procedure	Creates a repository for the primitive rule conditions that can be shared by multiple rules from the same or different rule classes
CREATE_EVENT_STRUCT procedure	Creates an event structure
CREATE_EXPFIL_INDEXES procedure	Creates expression filter indexes for the rule class if the default indexes have been dropped
CREATE_INTERFACE procedure	Creates a rule class interface package to directly operate on the rule class
CREATE_RULE_CLASS procedure	Creates a rule class
DELETE_RULE procedure	Deletes a rule from a rule class
DROP_CONDITIONS_TABLE procedure	Drops the conditions table

Table 8–1 (Cont.) DBMS_RLMGR Procedures

Procedure	Description
DROP_EVENT_STRUCT procedure	Drops an event structure
DROP_EXPFIL_INDEXES procedure	Drops Expression Filter indexes for the rule conditions
DROP_INTERFACE procedure	Drops the rule class interface package
DROP_RULE_CLASS procedure	Drops a rule class
EXTEND_EVENT_STRUCT	Adds an attribute to the primitive event structure
GET_AGGREGATE_VALUE function	Retrieves the aggregate value computed for a collection event
GRANT_PRIVILEGE procedure	Grants a privilege on a rule class to another user
PROCESS_RULES procedure	Processes the rules for a given event
PURGE_EVENTS procedure	Resets the rule class by removing all the events associated with the rule class and purging any state information pertaining to rules matching some events
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
SYNC_TEXT_INDEXES procedure	Synchronizes the indexes defined to process the predicates involving the CONTAINS operator in rule conditions

Rules Manager Views

Rules Manager defines views containing metadata using the `xxx_RLMGR` prefix, where `xxx` can be the string `USER` or `ALL`. Rules Manager procedures create and maintain these read-only views.

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

Table 9-1 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

9.1 USER_RLMGR_EVENT_STRUCTS View

The `USER_RLMGR_EVENT_STRUCTS` view lists all the event structures in the current schema. Table 9-2 describes the columns in this view.

Table 9-2 USER_RLMGR_EVENT_STRUCTS View

Column Name	Data Type	Description
<code>EVENT_STRUCTURE_NAME</code>	<code>VARCHAR2</code>	Name of the event structure
<code>HAS_TIMESTAMP</code>	<code>VARCHAR2</code>	Specifies whether the event structure has the event creation timestamp - YES/NO
<code>IS_PRIMITIVE</code>	<code>VARCHAR2</code>	Specifies whether the event structure is strictly primitive - YES/NO
<code>TABLE_ALIAS_OF</code>	<code>VARCHAR2</code>	Table name for a table alias primitive event
<code>CONDITIONS_TABLE</code>	<code>VARCHAR2</code>	Name of the table that stores the sharable conditions for this event structure

9.2 USER_RLMGR_RULE_CLASSES View

The `USER_RLMGR_RULE_CLASSES` view lists all the rule classes in the current schema. Table 9-3 describes the columns in this view.

Table 9–3 USER_RLMGR_RULE_CLASSES View

Column Name	Data Type	Description
RULE_CLASS_NAME	VARCHAR2	Name of the rule class
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 rule class is configured for composite events; if so, the value is YES
SEQUENCE_ENB	VARCHAR2	Indicates whether the rule class is enabled for rule conditions with sequencing; if so, the value is YES
AUTOCOMMIT	VARCHAR2	Indicates whether the rule 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
CNF_EVENTS	VARCHAR2	Types of continuous query notifications (formerly database change notifications) enabled for event management

9.3 USER_RLMGR_RULE_CLASS_STATUS View

The USER_RLMGR_RULE_CLASS_STATUS view lists the progress of rule class creation. [Table 9–4](#) describes the columns in this view.

Table 9–4 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

9.4 USER_RLMGR_PRIVILEGES View

The USER_RLMGR_PRIVILEGES view lists the privileges for the rule classes. [Table 9–5](#) describes the columns in this view.

Table 9–5 *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
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

9.5 USER_RLMGR_COMPRCLS_PROPERTIES View

The USER_RLMGR_COMPRCLS_PROPERTIES view lists the primitive events configured for a rule class and their properties. Table 9–6 describes the columns in this view.

Table 9–6 *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
COLLECTION_ENB	VARCHAR2	Is the primitive event enabled for collections?
GROUPBY_ATTRIBUTES	VARCHAR2	Event attributes that may be used for GROUPBY clauses

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`.

10.1 Law Enforcement Rules Application

This Law Enforcement Rules application defines rules to raise security alerts, place a person on the watch list, and so forth based on certain criteria. For this purpose, this application uses some real-world events such as bank transactions, transportation, and field reports 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 uses a set of predefined columns to store the rule identifiers, rule conditions and the descriptions. In addition to these columns, this rule class table defines two columns, `actionType` and `actionParam`, as specified through the `actprf_spec` argument. These columns capture the type of action that should be carried out 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)
RLM\$ENABLED		CHAR(1) DEFAULT 'Y'

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 event instances that match the rule and the action preferences associated with the rule. For this use case, the callback procedure considers the action for the rules to be the insertion of a detailed message into the message queue table. 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 you define in the rule 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 rule specifies the join predicate 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 the rule meets two of the following three conditions. The person gets a money transfer for over \$10,000 from outside the United States, he rents a truck, one-way, into one of the restricted areas, and he makes a phone conversation with a person already on

the watch list. The following rule demonstrates the use of the <ANY> element where it considers a rule condition to be 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 it considers a rule condition to be true if some of the specified events are detected and 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 receives over \$10,000 from outside the United States, he rents a truck for a one way trip into a restricted area, and a field report saying that the subject was never arrested before was not submitted within a "certain" (0.001 fraction of a day; this could be days, but it uses seconds 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 it executes the rules action (it prints the message to the screen as well as inserts it 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. Rule 4 executes the following action after the deadline time elapses:

```
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]

```

10.2 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. You cannot create the event structures from object types if they refer to the existing tables using table alias constructs. Instead, model such event structures as Expression Filter attribute sets, as follows:

```

begin
  DBMS_RLMGR.CREATE_EVENT_STRUCT (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_viewmm => '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
RLM\$ENABLED	CHAR(1) DEFAULT 'Y'	IN

4. Implement the callback procedure to perform the appropriate action for each matching rule, based on event instances that match the rule and action preferences associated with the rule. In this case, consider one action to be a message displayed to the screen, 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 1: If the order is for more than 100 routers and payment is by check, contact the customer to update the status of the order. Note that the rule specifies the join predicate across event types 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 2: If a Gold customer places an order and items ship before receiving 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 3: If a Gold customer places an order and items ship 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 rules for some primitive events by inserting rows 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 rule 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 the rules do not configure DML events for this rule class, the application must explicitly process the rules for the rows in the data table. To process the rules, it uses the ROWIDS of the rows inserted into the data tables as references to the events and passes these ROWIDS to the PROCESS_RULES procedure.

```

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)
RLM\$ENABLED		CHAR (1) DEFAULT 'Y'

```

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 the rule class results view stores the results from the matching of events with the rules. 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 table aliases configure the event structure, ROWIDS from the corresponding tables represent events, 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	
-----			-----
AAA0BxAAEAAAAHPAAC	AAA0ByAAEAAAAHXAAAC		2
UPDATE_CUST_PROFILE	DECR_AVAILABLE_CREDIT		
AAA0BxAAEAAAAHPAAC	AAA0ByAAEAAAAHXAAAC		3
UPDATE_STATISTICS	INCREMENT_QOS		
AAA0BxAAEAAAAHPAAC	AAA0BzAAEAAAAHfAAC		1
CALL_CUSTOMER	UPDATE_ORDER_STATUS		

4. Use ROWIDS 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

10.3 Use of Collections in an Order Management Application

The following Order Management application demonstrates the use of collection events for identifying complex event scenarios and acting on them. This application uses the object types in the database as the event structure and the basic steps in creating this application are similar to those discussed in [Section 10.1](#).

1. Create the object types that represent the primitive event structures and the composite event structure.

```
create or replace type PurchaseOrder as object
(  orderid      number,
   customerid   number,
   itemid       number,
   itemcount    number,
   amount       number,
   exptddate    date);
/

create or replace type ShipItem as object
(   itemid      number,
   itemtype     varchar2(30),
   orderid      number,
   truckid      number);
/

create or replace type TruckAtDock as object
(   truckid     number,
   loadid       date,
   status       varchar2(30),
   capacity     number);
/

create or replace type OrderMgmt as object
(
   porder PurchaseOrder,
   sitem  ShipItem,
   truck  TruckAtDock
);
/
```

2. Create the rule class. Set the rule class properties such that events based on PurchaseOrder and ShipItem types are enabled for collections.

```
BEGIN
  DBMS_RLMGR.CREATE_RULE_CLASS(
    rule_class      => 'OrderMgmtRC',
    event_struct    => 'OrderMgmt',
    action_cbk      => 'OrderMgmtCBK',
    actprf_spec     => 'actionType VARCHAR2(40),
                       actionParam VARCHAR2(100),
                       poAggrRet VARCHAR2(20) default null',
    rslt_viewnm     => 'MatchedScenarios',
    rlcls_prop      =>
      '<composite
        equal="(porder.orderid, sitem.orderid) |
              (sitem.truckid, truck.truckid)"
        ordering="rlm$rule.rlm$ruleid, porder.orderid,
              porder.itemid, truck.loadid">
        <collection type="PurchaseOrder"
          groupby="orderid, customerid, itemid"/>
        <collection type="ShipItem"
          groupby="itemid, truckid"/>
      </composite>');
END;
/
```

3. Implement the action callback procedure. Note that for each primitive event type enabled for collections, the action callback procedure uses one additional ROWID argument that binds in the identifier for the collection event. It then uses this event identifier to obtain any aggregate values computed for a given rule.

```

create or replace procedure "ORDERMGMTCBK" (
  PORDER      PURCHASEORDER,
  PO_EVTID    ROWID,
  SITEM       SHIPITEM,
  SI_EVTID    ROWID,
  TRUCK       TRUCKATDOCK,
  rlm$rule    ORDERMGMTRC%ROWTYPE) is
  mesg        VARCHAR2(100);
  aggrval     VARCHAR2(100);
begin
  mesg := ' Rule "' || rlm$rule.rlm$ruleid ||
          '" matched ' ||
          case when porder.orderid is not null then 'Purchase Order'
              || porder.orderid
              when porder.customerid is not null then 'Customer'
              || porder.customerid
              when sitem.truckid is not null then '||Truck ' || sitem.truckid
          end;
  if (porder is not null and rlm$rule.poAggrRet is not null) then
    aggrval := dbms_rlmgr.get_aggregate_value ('OrderMgmtRC', po_evtid,
                                              rlm$rule.poAggrRet);
    aggrval := ' with ' || rlm$rule.poAggrRet || ' equal to ' || aggrval;
  end if;
  dbms_output.put_line (mesg || aggrval);
end;
/

```

4. Create any user-defined function that may be used in the rule class.

```

create or replace function CustomerType (custId int) return VARCHAR2 is
begin
  return 'GOLD';
end;
/
exec dbms_rlmgr.add_functions('OrderMgmt', 'CustomerType');

```

5. Add rules to the rule class.

- a. Rule: Offer an elite status to a customer if he submits a large number of orders, each with a minimum of 10000 dollars.

```

insert into OrderMgmtRC (rlm$ruleid, actionType, actionParam,
                        rlm$ruledesc, rlm$rulecond) values
('Large number of orders promo', 'PROMOTION', 'ELITE_STATUS',
 'Offer an elite status to a customer if he submitted a large number
 of orders, each with a minimum of 10000 dollars',
 '<condition>
  <collection name="porder" groupby="customerid"
    having="count(*) > 10">
    amount > 10000
  </collection>
</condition>');

```

- b. Rule: Offer a promotion for ordering in bulk if the average value of the last 10 orders is over 20000 dollars.

```

insert into OrderMgmtRC (rlm$ruleid, actionType, actionParam,

```

```

                rlm$ruledesc, rlm$rulecond) values
('Expanding customer', 'PROMOTION', 'LARGE_ORDER',
'Offer a promotion for ordering in bulk if the average size of the
last 10 orders is over 20000 dollars',
'<condition>
  <collection name="porder" groupby="customerid"
                windowsize="10"
                having="avg(amount) > 20000"/>
</condition>');

```

- c. Rule: Offer an elite status to a customer if he submits a large number of orders, each with a minimum of 1000 dollars, within a 30 day period.**

```

insert into OrderMgmtRC (rlm$ruleid, actionType, actionParam,
                        rlm$ruledesc, rlm$rulecond) values
('Promo on Total size of orders in 10 days ', 'PROMOTION', 'ELITE_STATUS',
'Offer an elite status to a customer if he submitted a large number
of orders, each with a minimum of 1000 dollars, in a 30 day period',
'<condition>
  <collection name="porder" groupby="customerid"
                windowlen="30"
                having="sum(amount) > 50000"/>
        amount > 1000
  </collection>
</condition>');

```

- d. Rule: Compare the number of items ordered and the items shipped to mark the order complete.**

```

insert into OrderMgmtRC (rlm$ruleid, actionType, actionParam,
                        rlm$ruledesc, rlm$rulecond) values
('Completed order', 'UPDATE_ORDER_STATUS', 'COMPLETE',
'Compare the number of items ordered and the items shipped to mark the
order complete',
'<condition>
  <and equal="porder.orderid, sitem.orderid"
        having="count(sitem.*) = porder.itemcount">
    <object name="porder"/>
    <collection name="sitem" groupby="orderid" compute="count(*)">
      itemtype != 'Reusable Container'
    </collection>
  </and>
</condition>');

```

- e. Rule: Signal readiness to ship when the truck is at least 90% full.**

```

insert into OrderMgmtRC (rlm$ruleid, actionType, actionParam,
                        rlm$ruledesc, rlm$rulecond) values
('Ready to ship', 'READY_TO_SHIP', 'LOADED_TRUCK',
'Signal readiness to ship when the truck is at least 90% full',
'<condition>
  <and equal="sitem.truckid, truck.truckid"
        having="count(sitem.*) >= truck.capacity*0.9" >
    <object name="truck"> status = 'Loading' </object>
    <collection name="sitem" groupby="truckid" compute="count(*)">
      itemtype = 'Reusable Container'
    </collection>
  </and>
</condition>');

```

6. Process the rules for the instances of `PurchaseOrder`, `ShipItem`, and `TruckAtDock` events.

Part II

Expression Filter

This part introduces developing applications using Expression Filter feature.

Part II contains the following chapters:

- [Chapter 11, "Oracle Expression Filter Concepts"](#)
- [Chapter 12, "Indexing Expressions"](#)
- [Chapter 13, "Expressions with XPath Predicates"](#)
- [Chapter 14, "Expressions with Spatial and Text Predicates"](#)
- [Chapter 16, "Using Expression Filter with Utilities"](#)
- [Chapter 15, "SQL Operators and Statements"](#)
- [Chapter 17, "Object Types"](#)
- [Chapter 18, "Management Procedures Using the DBMS_EXPFIL Package"](#)
- [Chapter 19, "Expression Filter Views"](#)

Oracle Expression Filter Concepts

Oracle Expression Filter, a feature beginning in 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.

11.1 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.

11.1.1 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. You include data about cars for sale 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. You can store data in a table called `CARS` and match it 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, you store incoming data about cars 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 you make changes to the data or the expressions. Compare this to the traditional approach where you create a mapping table to store the relationship between the two tables. You must define a trigger to recompute the relationships and to update the mapping table if the data or expressions change. In this case, the application must compare new data to all expressions, and compare a new expression 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).

11.2 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 Database, 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 Language 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 Database and captures the interest of a customer 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 11.2.2](#).) 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.

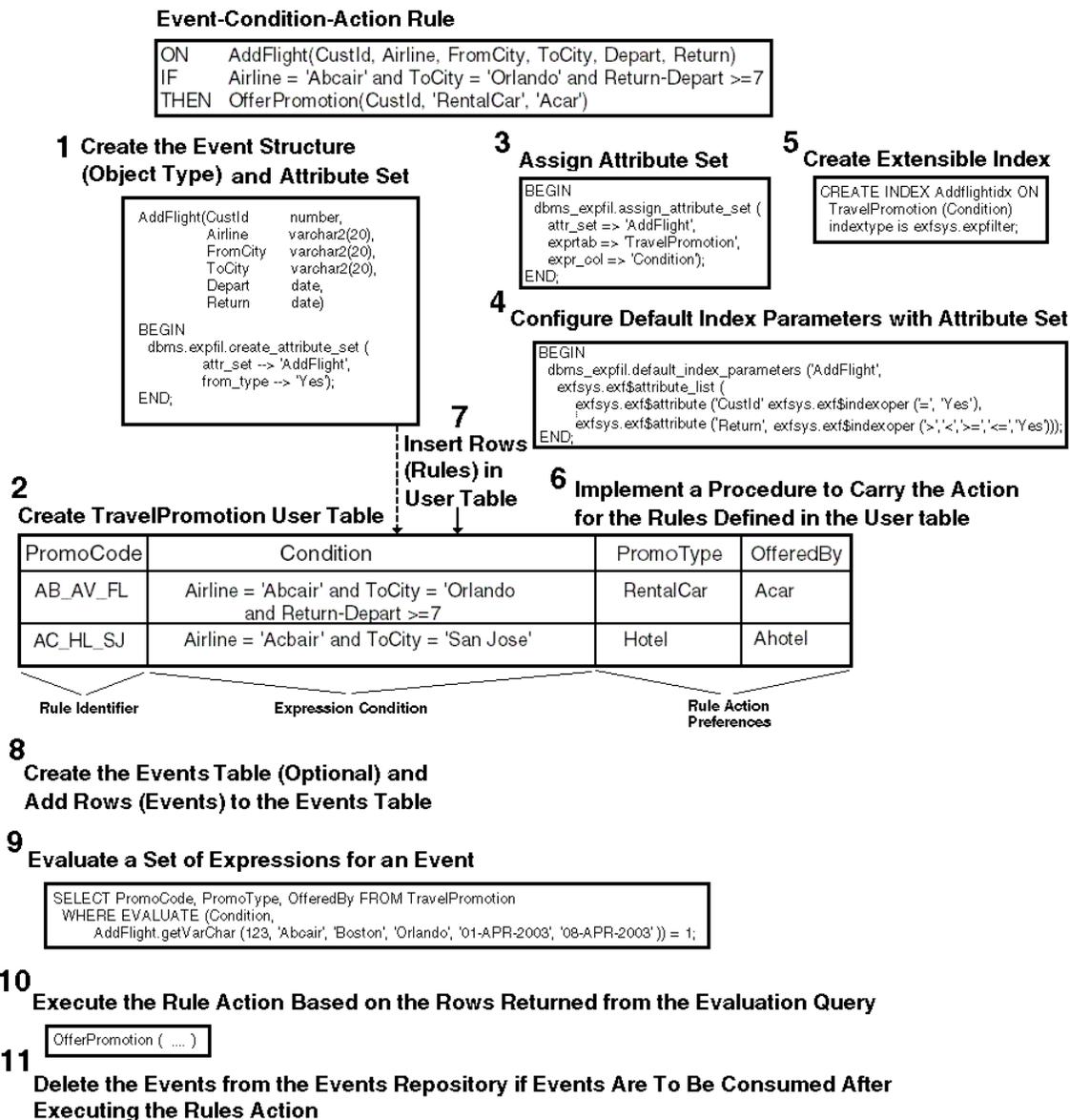
There are four basic steps to create and use an Expression column:

1. Define an attribute set. See [Section 11.2.1](#).
2. Define an Expression column in a user table. See [Section 11.2.2](#).

3. Insert expressions in the table. See [Section 11.2.3](#).
4. Apply the SQL EVALUATE operator to compare expressions to incoming data items. See [Section 11.3](#).

Figure 11–1 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 11–1 Expression Filter Implementation Process for a Rules Application



11.2.1 Defining Attribute Sets

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

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 11-1](#).
- 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 11-2](#).

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 11-3](#).

Attribute Set Examples

[Example 11-1](#) shows how to use an existing object type to create an attribute set. It uses the `CREATE_ATTRIBUTE_SET` procedure.

Example 11-1 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 11-2](#) 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 11-2 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)');
END;
```

```

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 11-3](#) shows how to add user-defined functions, using the `ADD_FUNCTIONS` procedure, to an attribute set.

Example 11-3 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`.

11.2.2 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 you can use in the expression set. You can use an attribute set 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. You can also use an existing `VARCHAR2` column in a user table for this purpose. The following example creates a table with a `VARCHAR2` column, named `Interest`, that will be used with an attribute set:

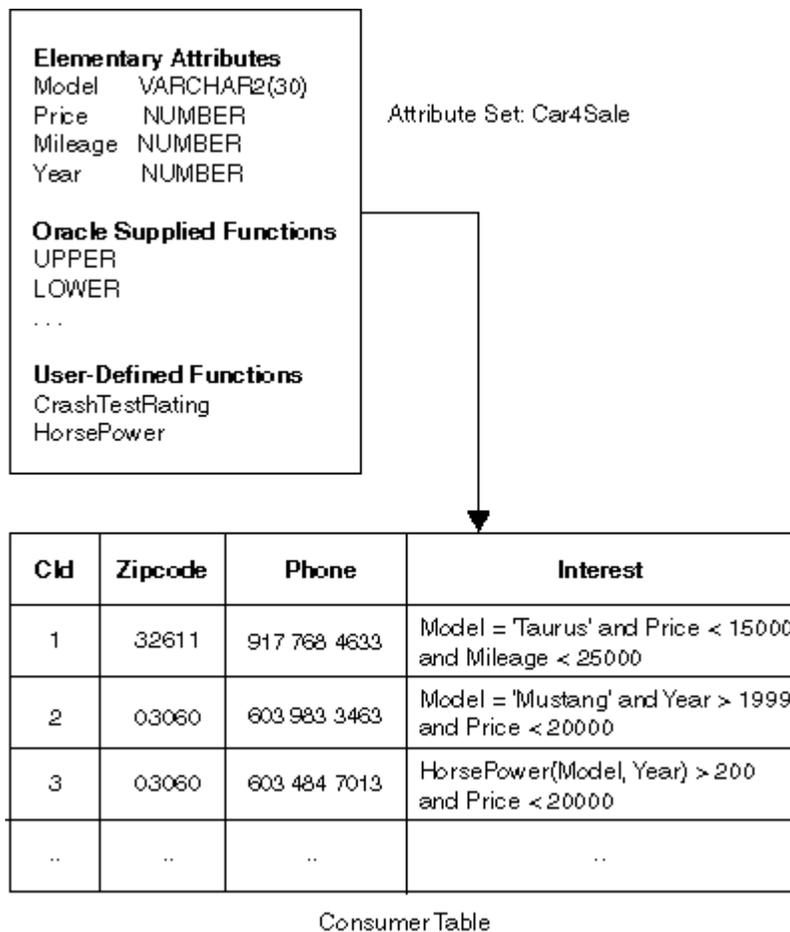
```
CREATE TABLE Consumer (CId          NUMBER,
                       Zipcode      NUMBER,
                       Phone         VARCHAR2(12),
                       Interest      VARCHAR2(200));
```

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 11-2](#) is a conceptual image of consumers' interests (in trading cars) being captured in a `Consumer` table.

Figure 11–2 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.

11.2.3 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. If the expression is not correct, SQL returns an error message. For more information about evaluation semantics, see [Section 11.4](#).

[Example 11–4](#) shows how to insert an expression (the consumer's interest in trading cars, which is depicted in [Figure 11–2](#)) into the `Consumer` table using the SQL `INSERT` statement.

Example 11–4 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 11–3](#)). [Example 11–5](#) shows how to insert an expression with a reference to a user-defined function, `HorsePower`, into the `Consumer` table.

Example 11–5 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 16.1](#).

11.3 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, you must provide values for all elementary attributes in the attribute set associated with the Expression column. Null is an acceptable value. You can specify the data item 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 you represent the values of all the elementary attributes in the attribute set as readable values, such as those stored in `VARCHAR`, `DATE`, and `NUMBER` data types and the constructors formatted as a string, then you can format the data item 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 you provide 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. Use the `MEMBER` method if the object instance is already available.

Expression Filter implicitly creates the `STATIC` and `MEMBER` methods 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 Database that can hold instances of any Oracle data type, both supplied by Oracle Database and user-defined. For more information, see *Oracle Database Object-Relational Developer's Guide*.

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, you can pass the data item 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;
```

For the syntax of the SQL `EVALUATE` operator, see ["EVALUATE" in Chapter 15](#). For additional examples of the SQL `EVALUATE` operator, see [Appendix B](#).

11.4 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.

Use the Expression Validation utility 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.

11.5 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. Other users 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.

11.6 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.

You can define an index 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. Expression Filter uses the indextype, `EXFSYS.EXPFILTER`, 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 12.3](#)), index processing ([Section 12.4](#)), and user commands for creating and tuning indexes ([Section 12.6](#)).

12.1 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)`.

12.2 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.

12.3 Index Representation

The Expression Filter index uses persistent database objects internally to maintain the index information for an expression set. A relational table called the predicate table captures the grouping information for all the predicates in an expression set. 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 12-1](#).

Figure 12–1 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, Expression Filter stores its operator and the right-hand side constant 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 Expression Filter stores them in a VARCHAR2 column of the predicate table as sparse predicates. (For the example in [Figure 12–1](#), the predicates on Mileage and Year fall in this category.) Expression Filter implicitly treats the predicates with IN lists and the predicates with a varying right-hand side (not a constant) as sparse predicates. Expression Filter creates native indexes on the predicate table as described in [Section 12.4](#).

12.4 Index Processing

To evaluate a data item for a set of expressions, Expression Filter computes the left-hand side of each predicate group in the data item and compares its value 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 12–1](#)), 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. Expression Filter can configure duplicate predicate groups 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, Expression Filter evaluates the corresponding sparse predicates that are stored in the predicate table 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), Expression Filter creates concatenated bitmap indexes 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, Expression Filter divides the predicates from an expression set into three classes:

1. **Indexed predicates:** Predicates that belong to a subset of the preconfigured predicate groups that are identified as most discriminating. Expression Filter creates bitmap indexes 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. Expression Filter captures these predicates 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 Expression Filter creates bitmap indexes 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, Expression Filter evaluates last sparse predicates (if any). If the expressions with sparse predicates evaluate to true, Expression Filter considers them 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 you can configure the Expression Filter index to check only for equality predicates while processing the indexed predicate groups. Expression Filter processes and evaluates sparse predicates along with any other form of predicate on the `Model` attribute at the same time.

12.5 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. Expression Filter uses the same query (with bind variables) 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.

12.6 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-tunes 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 13](#) 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](#).

12.7 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 (10.2), 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.

12.8 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 chapter describes an application for XPath predicates using the `Car4Sale` example introduced in [Chapter 11](#). 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
```

You can combine this predicate with one or more predicates on other XML or non-XML attributes.

13.1 Using XPath Predicates in Expressions

Using the `XMLType` data type supplied by Oracle Database, you 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 `EXISTSNODE` on an instance of the `XMLType` data type to process an XPath expression for the XML instance. For more information, see *Oracle Database SQL Language Reference* and *Oracle XML DB Developer's Guide*.

To allow XPath predicates in an expression set, you should create the corresponding attribute set 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, you can process a set of expressions stored in the Interest column of the Consumer table 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.

13.2 Indexing XPath Predicates

To process a large set of XPath predicates in an expression set efficiently, you can configure the Expression Filter index defined for the expression set 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, Expression Filter treats an XPath predicate 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`.

13.2.1 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 12.4](#).

You can configure a positional filter for an Expression Filter index 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, the following XPath predicate can be indexed, but a similar predicate with an `IS NULL` check on the return value cannot be indexed.

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

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.

13.2.2 Index Representation

You can configure the Expression Filter index 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 12.3](#) 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 13–1](#) 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 12–1](#). 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 13–1 Conceptual Predicate Table with XPath Predicates

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

13.2.3 Index Processing

The XPath predicates that Expression Filter captures 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 13–1](#)), 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, you can represent the values for all the occurrences of the `stereo@make` attribute in the given XML document as a VARRAY and these 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` variable 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. Expression Filter identifies a subset of the XML tags as the most selective predicate groups and configures them as the indexed predicate groups (See [Section 12.4](#)). Expression Filter creates bitmap indexes 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.

13.2.4 Index Tuning for XPath Predicates

Expression Filter classifies the most discriminating XML tags in a set of XPath predicates 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.

You can assign a few XPath index parameters 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, Expression Filter uses the previous XPath index parameters 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 you create the index. 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 you create the index on a column storing the expressions, a query with the `EVALUATE` operator can efficiently process a large set of XPath and non-XPath predicates for a data item:

```

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;
```

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

Expressions with Spatial and Text Predicates

This chapter describes using expressions with spatial predicates and text predicates. The expression stored in the columns of Expression data type can specify predicates involving:

- Spatial predicates on SDO_GEOMETRY attributes (see [Section 14.1](#))
- Text predicates on Text data (see [Section 14.2](#))

14.1 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 11](#). 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. You can specify this 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 14.1.1](#) and [Section 14.1.2](#) describes how to specify spatial predicates in arbitrary expressions and how to ensure the predicates are evaluated using appropriate spatial indexes.

14.1.1 Using Spatial Predicates in Expressions

Using the Oracle Database supplied `SDO_GEOMETRY` data type, you 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 Developer's Guide*.

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, you should associate the geometry metadata describing the dimension, lower and upper bounds, and tolerance in each dimension 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 Developer's Guide*.

```
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.

14.1.2 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 the 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.

14.2 Expressions with Text Predicates

The Text predicates in the stored expressions are specified using the `CONTAINS` operator within the SQL `WHERE` clause syntax. The text predicates include the document to be searched (through an attribute name) and a text query expression. The text query expression specifies the text pattern to be searched within the document and it is represented using the `CONTEXT` grammar (see *Oracle Text Application Developer's Guide*).

```

Model='Taurus' and Price < 15000 and Mileage < 25000 AND
  CONTAINS (InsReport, 'near(water, damage), 4') = 1

```

The Text classification engine currently allows storing (text) query expressions in a column of a table and processing them for incoming documents. The incoming documents are matched with the stored expressions using a `MATCHES` operator within SQL queries. The metadata required to match the document with the stored expressions is obtained from the `CTXRULE` index defined on the column storing the query expressions. This metadata includes the information about the text preferences such as the type of `LEXER` to use and the list of `STOP` words. This metadata is assigned to the `CTXRULE` index at the time of index creation and hence the `MATCHES` operator is only operational when the `CTXRULE` index exists on the column.

The support for text predicates in stored expressions is provided through the integration of Text classification engine and the Expression Filter feature. In these

stored expressions, the text predicates can be combined with other text predicates, scalar or XML predicates using the SQL `WHERE` clause syntax. You specify the text predicates using the `CONTAINS` operator, which identifies the text attribute on which the text query expression should be evaluated. At the time of expression evaluation, Expression Filter matches the text query expression with the document assigned to the text attribute. The text preferences required for this matching operation are assigned to the text attributes as a form of metadata.

In an attribute set you create for an expression column, a text attribute is identified as an attribute of `CLOB` or `VARCHAR` data type with associated text preferences. You use such attributes as the first argument to the `CONTAINS` operator to form text predicates within the stored expressions. You specify the text preferences for an attribute using an instance of `EXF$TEXT` type, which accepts the text preferences in string format, for example: `(LEXER insrpt_lexer WORDLIST insrpt_wordlist)`. The preferences you specify through this argument are used for parsing the document bound to the attribute and for indexing the text query expressions (using `ctxsys.CTXRULE` Indextype). Alternately, you can assign an `EXF$TEXT` instance with an empty preferences string to use default preferences for a `CTXRULE` index (See *Oracle Text Reference*).

```
begin
  dbms_expfil.create_attribute_set(attr_set => 'Car4Sale');

  // create scalar attributes
  dbms_expfil.add_elementary_attribute(
    attr_set => 'Car4Sale',
    attr_name => 'Model',
    attr_type => 'VARCHAR2(30)');

  dbms_expfil.add_elementary_attribute(
    attr_set => 'Car4Sale',
    attr_name => 'Price',
    attr_type => 'NUMBER');

  . . .
  // create text attribute
  dbms_expfil.add_elementary_attribute(
    attr_set => 'Car4Sale',
    attr_name => 'InsReport',
    attr_type => 'CLOB'
    attr_type =>
      exf$text(
        'LEXER insrpt_lexer
        WORDLIST insrpt_wordlist'));
end;
```

Note: The attribute sets consisting of one or more text attributes have to be assembled using the `ADD_ELEMENTARY_ATTRIBUTE` procedure and they cannot be created from an existing object type.

For the attribute set previously described, you should create the `LEXER` preference `insrpt_lexer` and the `WORDLIST` preference `insrpt_wordlist` using the `CTX_DDL` package (see *Oracle Text Reference*).

Figure 14–1 Text Predicate in the Stored Expression Using the CONTAINS Operator

Car4Sale (Model VARCHAR(10), Year NUMBER, Mileage NUMBER, Price NUMBER, InsReport CLOB extText('LEXER ..'))		Attribute Set – Car4Sale		
CId	Interest	Zipcode	CR	..
1	Model = 'Taurus' and Price < 15000 and Mileage < 25000 and CONTAINS(InsReport, 'near((water,damage),4)') = 1	32611	2	..
2	Model = 'Mustang' and Year > 1999 and Price < 20000 and CONTAINS(InsReport, 'crash test rating') = 1	03060	4	..
3	Model = 'Mustang' and Year < 1970
..

When you use the attribute set with a text attribute as the metadata for an expression column, the expressions stored in the column can include text predicates. The text query expression you use within the text predicates use a subset of the `CONTEXT` grammar to support proximity searches, theme searches, and so forth, with the use of `CONTEXT` operators: `ABOUT`, `AND`, `NEAR`, `NOT`, `OR`, `STEM`, `WITHIN`, and `THESAURUS`. (See the section about document classification in *Oracle Text Application Developer's Guide* for complete syntax). Expression Filter validates the scalar predicates in the stored expressions for syntactic and semantic correctness when the expressions are inserted into a column of expression data type. However, the text query expressions you specify as the second argument to the `CONTAINS` operator are not validated until the time of index creation or index maintenance. Any text query expression errors identified during index creation are reported through `CTX_USER_INDEX_ERRORS` view as described next.

Unlike in the case of expressions containing purely scalar predicates, the expressions containing predicates on text attributes cannot be evaluated when the Expression Filter index is not defined on the column storing expressions. At the time of index creation, all the text attributes are added to the indexed predicate groups and `CTXRULE` indexes are created on the predicate table for each of these predicate groups. The `CTXRULE` index creation for the text predicate groups also identifies any errors in the corresponding text query expressions and these errors are reported in the `CTX_USER_INDEX_ERRORS` view (see *Oracle Text Reference*). However, the errors reported in this view refer to the rows in the predicate table structure of the Expression Filter index. These errors can be mapped back to the rows in the user table using the `USER_EXPFIL_TEXT_INDEX_ERRORS` view.

Once the index is defined, you can evaluate the stored expressions using the `EVALUATE` operator in a SQL query. When you define the attribute set with one or more text attributes, the data item passed to the `EVALUATE` operator consists of corresponding values (`VARCHAR` or `CLOB`) assigned to these attributes.

```
SELECT * FROM Consumer
WHERE EVALUATE (Interest,
  AnyData.convertObject (
    Car4Sale('Mustang',19000,25000,2001,
      '...4 star crash test rating ...') = 1
```

The previous query with the `EVALUATE` operator identifies all the expressions that are true based on the text and the non-text predicates.

When you define an Expression Filter index on the column storing expressions, some of the most common and selective scalar and XPath predicates are identified by the end-user and the predicate table is created to accommodate such predicates. While selecting the predicate groups for an Expression Filter index, the predicates involving a text attribute are always included in the indexed predicate groups. The predicate table is created with a `VARCHAR` column to store the text query expressions specified for a text attribute and a `CTXRULE` index is defined on this column. These indexes along with the bitmap indexes you define on the other indexed predicates process the expressions for a data item efficiently.

When you issue a query with the `EVALUATE` operator, the values from the data item passed in are bound into a query on the predicate table. The predicate table query uses a `MATCHES` operator on the columns storing text query expressions to match the incoming documents with the stored text query expressions. The `MATCHES` operator internally makes use of the `CTXRULE` index to process the text query expressions efficiently and return a subset of expressions that match the document. The results from matching the documents with text query expressions are combined with results from matching other scalar and XPath predicates to narrow down the working set to a set of candidate expressions that are true based on all indexed predicate groups. The processing of stored predicate groups and the sparse predicates remain unchanged with the inclusion of text predicates in the expression set.

Delayed DML Maintenance for Text Predicates

The modifications made to the data stored in Expression data type column are all transactional in nature. That is, any modifications made to the expression column are automatically reflected in the corresponding Expression Filter index. However, the modifications made to the text query expression within the stored expressions are not immediately reflected in the corresponding `CTXRULE` index. The index maintenance is forced by manually synchronizing the index, and until the `CTXRULE` index is synchronized, the Expression Filter index may return incorrect results (based on the old text predicates and new scalar predicates). One or more `CTXRULE` indexes defined for expression sets stored in a table can all be synchronized using a `DBMS_EXPFIL.SYNC_TEXT_INDEXES` procedure call. The name of the user table with one or more expression columns is passed as the only argument to this procedure call.

```
begin
  dbms_expfil.sync_text_indexes (expr_tab => 'Consumer');
end;
```

You must have the `EXECUTE` privilege on the `CTX_DDL` package for successful completion of the previous command. The call to the `DBMS_EXPFIL.SYNC_TEXT_INDEXES` procedure processes all the newly added or modified text predicates in the expression set and synchronizes the `CTXRULE` indexes accordingly. Any text predicate errors identified in this process are reported through the `USER_EXPFIL_TEXT_INDEX_ERRORS` view.

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 Language Reference*.

Table 15–1 *Expression Filter Index Creation and Usage Statements*

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

You can use the `EVALUATE` operator in the `WHERE` clause of a SQL statement. When you define an Expression Filter index 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. You can use the `EVALUATE` operator as a join predicate between a table storing expressions and a table storing the corresponding data items.

If you can represent the values of all elementary attributes in the attribute set as readable values, such as those stored in `VARCHAR`, `DATE`, and `NUMBER` data types and the constructors formatted as a string, then you can format the data item 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 you can format a list of values provided for the data item 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.

You can format any data item 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 11.3](#) for more information about the `EVALUATE` operator.

Related views: [USER_EXPFIL_ATTRIBUTE_SETS View](#), [USER_EXPFIL_ATTRIBUTES View](#), and [USER_EXPFIL_EXPRESSION_SETS View](#)

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 you issue the `ALTER INDEX . . . REBUILD` statement without a `PARAMETERS` clause, the Expression Filter index is rebuilt using the current list of stored and indexed attributes. You can also use this statement for indexes that failed during `IMPORT` operation due to missing dependent objects.

You can modify the default index parameters associated with an attribute set without affecting the existing Expression Filter indexes. You can rebuild these indexes to use the new set of defaults by using the `DEFAULT` parameter with the `ALTER INDEX . . . REBUILD` statement. When you rebuild an index using the defaults, index parameters assigned to the expression set are cleared.

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 12.8](#) for more information about rebuilding indexes.

Related views: [USER_EXPFIL_INDEXES View](#) and [USER_EXPFIL_PREDTAB_ATTRIBUTES View](#)

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. You should configure the column being indexed 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 you specify this parameter, 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 you specify this parameter, 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

You can use this parameter 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 Language 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 18](#) and the ["USER_EXPFIL_INDEX_PARAMS View"](#) in [Chapter 19](#).)

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 you do not specify the REPLACE DEFAULTS clause, this list is merged with the default index parameters associated with the corresponding attribute set.

If you do not specify the REPLACE DEFAULTS clause, 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 you use the TOP clause for the STOREATTRS parameter, the INDEXATTRS parameter (if specified) should also use the TOP clause. Also, the number you specify for the TOP clause of the INDEXATTRS parameter should be less than or equal to the one you specify for the STOREATTRS parameter. When you use a TOP clause, 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. You can specify an alternate tablespace and other storage parameters for the predicate table using the PREDSTORAGE clause. The indexes on the predicate table are always created in the same tablespace as the predicate table.

See [Chapter 12](#) for information about indexing expressions.

Related views: [USER_EXPFIL_INDEXES View](#), [USER_EXPFIL_INDEX_PARAMS View](#), [USER_EXPFIL_DEF_INDEX_PARAMS View](#), [USER_EXPFIL_EXPRSET_STATS View](#), and [USER_EXPFIL_PREDTAB_ATTRIBUTES View](#)

Examples

When you do not directly assign index parameters 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)');
```

You can use the TOP clause 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;
```

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.

16.1 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, you can validate the expressions in the table 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.

16.2 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.

16.2.1 Exporting and Importing Tables Containing Expression Columns

The following guidelines and known behavior associated with exporting and importing tables containing Expression columns can 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.

16.2.2 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.

16.2.3 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.

Object Types

The Expression Filter feature supplies 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. Use the `EXF$TABLE_ALIAS` type to support expressions defined on one or more database tables. [Table 17-1](#) 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, enclose the values with double quotation marks.

Tip: See the chapter describing the Expression Filter Object Types in *Oracle Database PL/SQL Packages and Types Reference* for all reference information concerning Expression Filter object types. Use the links in the Object Type Name column in [Table 17-1](#) to see each object type.

Table 17-1 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\$TEXT</code>	Associates preferences to a text attribute in an attribute set or an event structure.
<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 you can use to manage attribute sets, expression sets, expression indexes, optimizer statistics, and privileges. [Table 18–1](#) 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, enclose the values in double quotation marks.

Tip: See the chapter on the DBMS_EXPFIL package in *Oracle Database PL/SQL Packages and Types Reference* for all reference information concerning Expression Filter package procedures. Use the links in the Procedure column in [Table 18–1](#) to see each procedure.

Table 18–1 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 18–1 (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
SYNC_TEXT_INDEXES	Synchronizes the indexes defined to process the predicates involving the CONTAINS operator in stored expressions
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

Expression Filter defines views containing metadata using the `xxx_EXPFIL` prefix, where `xxx` can be the string `USER` or `ALL`. Expression Filter procedures create and maintain these read-only views.

Table 19–1 lists the names of the views and their descriptions.

Table 19–1 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
USER_EXPFIL_TEXT_INDEX_ERRORS	Maps any errors with the text indexes to the expression column values in which the error exists

19.1 USER_EXPFIL_ASET_FUNCTIONS View

The `USER_EXPFIL_ASET_FUNCTIONS` view lists all the functions and packages that are allowed in the expressions using a particular attribute set. The following table describes the columns in this view:

Column Name	Data Type	Description
<code>ATTRIBUTE_SET_NAME</code>	<code>VARCHAR2</code>	Name of the attribute set
<code>UDF_NAME</code>	<code>VARCHAR2</code>	Name of the user-defined function or package (or type) as specified by the user (with or without the schema extension)
<code>OBJECT_OWNER</code>	<code>VARCHAR2</code>	Owner of the function or package (or type)
<code>OBJECT_NAME</code>	<code>VARCHAR2</code>	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

19.2 USER_EXPFIL_ATTRIBUTES View

The USER_EXPFIL_ATTRIBUTES view lists all the elementary attributes of the attribute sets defined in the user's schema. The following table describes the columns in this view:

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
TEXT_PREFERENCES	VARCHAR2	Preferences for an attribute configured for text predicates

19.3 USER_EXPFIL_ATTRIBUTE_SETS View

The USER_EXPFIL_ATTRIBUTE_SETS view lists the attribute sets defined in the user's schema. The following table describes the columns in this view:

Column Name	Data Type	Description
ATTRIBUTE_SET_NAME	VARCHAR2	Name of the attribute set

19.4 USER_EXPFIL_DEF_INDEX_PARAMS View

The USER_EXPFIL_DEF_INDEX_PARAMS view lists the default index parameters (stored and indexed attributes) associated with the attribute sets defined in the user's schema. The following table describes the columns in this view:

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

Column Name	Data Type	Description
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

19.5 USER_EXPFIL_EXPRESSION_SETS View

The USER_EXPFIL_EXPRESSION_SETS view lists the expression sets defined in the user's schema. The following table describes the columns in this view:

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)

19.6 USER_EXPFIL_EXPRSET_STATS View

The USER_EXPFIL_EXPRSET_STATS view lists the predicate statistics for the expression sets in the user's schema. The following table describes the columns in this view:

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

Column Name	Data Type	Description
PCT_GT_OPER	NUMBER	Percentage of predicates (of the attribute) with the greater than (>) operator
PCT_LTEQ_OPER	NUMBER	Percentage of predicates (of the attribute) with the less than or equal to (<=) operator
PCT_GTEQ_OPER	NUMBER	Percentage of predicates (of the attribute) with the greater than or equal to (>=) operator
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

19.7 USER_EXPFIL_INDEX_PARAMS View

The USER_EXPFIL_INDEX_PARAMS view lists the index parameters associated with the expression sets defined in the user's schema. The following table describes the columns in this view:

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

19.8 USER_EXPFIL_INDEXES View

The USER_EXPFIL_INDEXES view lists the Expression Filter indexes defined in the user's schema. The following table describes the columns in this view:

Column Name	Data Type	Description
INDEX_NAME	VARCHAR2	Name of the index

Column Name	Data Type	Description
PREDICATE_TABLE	VARCHAR2	Name of the predicate table used for the index
ACCESS_FUNC_PACKAGE	VARCHAR2	Name of the package that defines the functions with queries on the predicate table
ATTRIBUTE_SET	VARCHAR2	Name of the corresponding attribute set
EXPRESSION_TABLE	VARCHAR2	Name of the table on which the index is defined
EXPRESSION_COLUMN	VARCHAR2	Name of the column on which the index is defined
STATUS	VARCHAR2	Index status: <ul style="list-style-type: none"> ▪ VALID: Index was created successfully ▪ FAILED: Index build failed, and it should be dropped and re-created ▪ FAILED REBLD: Index build or rebuild failed, and it can be rebuilt using the ALTER INDEX REBUILD statement

19.9 USER_EXPFIL_PREDTAB_ATTRIBUTES View

The USER_EXPFIL_PREDTAB_ATTRIBUTES view shows the exact list of stored and indexed attributes used for expression filter indexes in the user's schema. The following table describes the columns in this view:

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

19.10 USER_EXPFIL_PRIVILEGES View

The USER_EXPFIL_PRIVILEGES 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. The following table describes the columns in this view:

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

19.11 USER_EXPFIL_TEXT_INDEX_ERRORS

The USER_EXPFIL_TEXT_INDEX_ERRORS view maps any errors with the text indexes to the expression column values in which the error exists. The following table describes the columns in this view:

Column Name	Data Type	Description
EXPRESSION_TABLE	VARCHAR2	Table with the expression column
EXPRESSION_COLUMN	VARCHAR2	Name of the column storing expressions
ERR_TIMESTAMP	VARCHAR2	Time at which the error was noticed
ERR_EXPRKEY	VARCHAR2	Key to the expression with the text predicate
ERR_TEXT	VARCHAR2	Description of the text predicate error

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'
```

Create the attribute set for this type of expression 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. Configure the Expression column in this table 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 --
```

You can index the expressions that use one or more table alias attributes 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 you evaluate the expression, the values for the attributes defined as table aliases are passed by assigning the ROWIDs from the corresponding tables. You can evaluate the expressions stored in the `HRInterest` table 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;
```

You can add additional predicates 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 [Section 11.2.1](#). In this application, the `Consumer` table stores the information about consumers interested in buying used cars. In addition to the `Consumer` table described in [Section 11.2.2](#), 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;
```

You can also use the join semantics of the EVALUATE operator 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, you can capture the responsibilities of the representatives 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

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. Expression Filter creates objects with the EXF\$ prefix and Rules Manager creates objects with the RLM\$ prefix (). You can use Expression Filter or Rules Manager APIs to maintain these objects. 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 11](#):

```
-- 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 you create an Expression column by assigning an attribute set to a `VARCHAR2` column in a user table, a `BEFORE ROW` trigger is created on the table. Expression Filter uses this trigger to invoke the expression validation routines when a new expression is added or an existing expression is modified. Expression Filter always creates this trigger 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 12.8](#).

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 rule 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 EVALUATE 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 EVALUATE operator to compare rule conditions stored in the <code>rlm\$rulecond</code> column of the rule 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 beginning with 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 rule 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 rule 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 rule 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 rule 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 rule 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)
RLM\$ENABLED		CHAR(1) DEFAULT 'Y'

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

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

Once the rule 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 rule 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 rule 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 and higher. 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=".*"/>
      <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="cnfevents">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="I"/>
              <xsd:enumeration value="IU"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>

```

```

        <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:enumeration value="rule"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:attribute>
</xsd:restriction>
</xsd:complexContent>
</xsd:complexType>
<xsd:group name="ObjectOrCollectionProp">
    <xsd:choice>
        <xsd:element name="object" type="rlmp:PrimEventProp"
            minOccurs="0" maxOccurs="1"/>
        <xsd:element name="collection" type="rlmp:CollectionProp"
            minOccurs="0" maxOccurs="1"/>
    </xsd:choice>
</xsd:group>

<!-- Properties of a rule class with composite events -->
<xsd:complexType name="CompositeRuleSetProp">
    <xsd:sequence>
        <xsd:group ref="rlmp:ObjectOrCollectionProp" 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:complexType name="CollectionProp">
  <xsd:complexContent>
    <xsd:restriction base="xsd:anyType">
      <xsd:attribute name="type" type="xsd:string" use="required"/>
      <xsd:attribute name="groupby" type="xsd:string"/>
      <xsd:attribute name="compute" type="xsd:string"/>
    </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:unique>
  </xsd:element>
  <xsd:group name="ObjectOrCollectionCondition">
    <xsd:choice>
      <xsd:element name="object" type="rlmc:ObjectCondType"
        minOccurs="1" maxOccurs="1"/>
      <xsd:element name="collection" type="rlmc:CollectionCondType"
        minOccurs="1" maxOccurs="1"/>
    </xsd:choice>
  </xsd:group>

  <xsd:complexType name="AndType">
    <xsd:sequence>
      <xsd:group ref="rlmc:ObjectOrCollectionCondition"
        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="having" 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="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:complexType>

```

```

</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:attribute name="ref" type="xsd:string"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="CollectionCondType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:string">
      <xsd:attribute name="name" type="xsd:string" use="required"/>
      <xsd:attribute name="groupby" type="xsd:string" use="required"/>
      <xsd:attribute name="having" type="xsd:string"/>
      <xsd:attribute name="compute" type="xsd:string"/>
      <xsd:attribute name="windowsize" type="xsd:string"/>
      <xsd:attribute name="windowlen" type="xsd:string"/>
    </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*

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

If you implement a single PL/SQL procedure in the database 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 you do not implement the action for all the rules 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	rlm\$enabled	ActionCommands	rlm\$ruledesc
AB_AV_ORL	Airline='Abcair' and ToCity='Orlando'	'Y'	begin OfferAcarPromotion(:1,10); end;	Additional info
AC_HT_SJC	Airline='Acbaair' and ToCity='San Jose'	'Y'	begin OfferAhotelPromotion (:1, 5); end;	Additional info
...

For the preceding rule class configuration, you can implement the action callback procedure 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).

A

action callback procedure, 2-3
 implementing various forms of rule actions, G-1
ADD_ELEMENTARY_ATTRIBUTE procedure, 11-5
ADD_FUNCTIONS procedure, 11-6
ALTER INDEX REBUILD statement, 15-4
ALTER INDEX RENAME TO statement, 15-5
AnyData.convertObject method, 11-10, 15-2
application examples, B-1
 active application, B-1
 batch evaluation, B-2
 law enforcement, 10-1
 order management, 10-8
 resource management, B-2
ASSIGN_ATTRIBUTE_SET procedure, 11-7, 16-2
attribute sets
 automatically creating, 11-5
 copying, 11-8
 creating with an existing object type, 11-5
 dropping, 11-6, 11-8
 examples, 11-5
 unassigning, 11-8
autocommit, 3-9, 3-10

B

BUILD_EXCEPTIONS_TABLE procedure, 11-11
bulk loading, 16-1
 bypassing validation, 16-1
bypassing validation, 16-1

C

complex rule conditions, 2-12
composite event, 2-1, 2-7
 rule conditions with ANY n semantics, 5-10
 rule conditions with set semantics, 5-10
 sequencing of primitive events, 5-5
conditional expressions
 See expressions
conflict resolution, 3-3
consumption of events, 3-2
converting rules applications, D-3
COPY_ATTRIBUTE_SET procedure, 11-8
CREATE INDEX statement, 12-6, 15-6

CREATE_ATTRIBUTE_SET procedure, 11-5

D

data item
 formatted as AnyData, 11-10
 formatted as name/value pair, 11-10
 formatted as string, 11-9
database
 exporting, 16-3
DBMS_EXPFIL package, 11-3
 GRANT_PRIVILEGE procedure, 11-11
 methods, 18-1
 REVOKE_PRIVILEGE procedure, 11-11
DBMS_RLMGR package
 methods, 8-1
decision points in you application, 1-4
DEFAULT_INDEX_PARAMETERS procedure, 12-6
DEFAULT_XPINDEX_PARAMETERS
 procedure, 13-5
DEFRAG_INDEX procedure, 12-8
deinstall script
 Rules Manager and Expression Filter, E-1
developing rules applications
 decision points, 1-4
 using Rules Manager, 1-4
DML events, 3-11
DROP INDEX statement, 15-9
DROP_ATTRIBUTE_SET procedure, 11-6, 11-8
duration of primitive events, 3-5

E

ECA components
 defined, 1-2
ECA rules, 2-4
 defined, 1-2
elementary attributes, 11-3
equality join predicates, 3-6
error messages, 11-12
EVALUATE operator, 11-9, 15-2
 arguments, 11-9
evaluating composite events
 using complex rule conditions, 2-12
evaluating rules
 incremental, 3-5

- negation in rules condition, 5-6
- event, 2-1
 - composite, 2-7
 - primitive, 2-5, 2-13, 6-1
- event management policies, 2-12, 3-1
 - autocommit, 3-9, 3-10
 - conflict resolution, 3-3
 - consumption of events, 3-2
 - DML events, 3-11
 - duration of primitive events, 3-5
 - equality join predicates, 3-6
 - order of rule execution, 3-4
 - specifying storage properties for objects created for the rule class, 3-9
- event structure, 2-1
- EXF\$ATTRIBUTE object type, 17-1
- EXF\$ATTRIBUTE_LIST object type, 17-1
- EXF\$INDEXOPER object type, 17-1
- EXF\$TABLE_ALIAS object type, 17-1
- EXF\$VALIDATE_n namespace, C-2
- EXF\$XPATH_TAG object type, 17-1
- EXF\$XPATH_TAGS object type, 17-1
- exporting
 - databases, 16-3
 - tables, 16-2
 - users, 16-3
- Expression column, 11-3, 11-7
 - creating, 11-3
- Expression data type, 11-3, 11-6
 - creating a column of, 11-7
- Expression Filter
 - active application example, B-1
 - batch evaluation example, B-2
 - configuring to process XPath predicates, 13-4
 - internal objects, C-1
 - overview, 11-1
 - resource management example, B-2
 - system triggers, C-2
 - usage scenarios, 11-1
 - utilities, 16-1
- expression sets, 11-3
 - allowing XPath predicates in, 13-1
- Expression Validation utility, 11-11
- expressions, 11-3
 - defined on one or more tables, A-1
 - definition, 11-3
 - deleting, 11-8
 - indexing, 12-1
 - inserting, 11-8
 - updating, 11-8
 - valid, 11-3
 - with spatial predicates, 14-1
 - with XPath predicates, 13-1

F

- features
 - new, xiii
- fragmentation of indexes, 12-8
- functions

- adding to attribute sets, 11-6

G

- GET_EXPRSET_STATS procedure, 12-7
- getVarchar methods
 - MEMBER, 11-10, 15-2
 - STATIC, 11-10, 15-2
- GRANT_PRIVILEGE procedure, 11-11

I

- importing
 - tables, 16-2
- incremental evaluation of rules, 3-5, 5-1
 - complex rule application
 - XML tag extensions, 5-1
- INDEX_PARAMETERS procedure, 12-6, 13-6
- indexed predicates, 12-4
- indexes
 - creating for expression set, 12-6, 15-6
 - creating from default parameters, 12-6
 - creating from exact parameters, 12-6
 - creating from statistics, 12-7
 - defragmenting, 12-8
 - dropping, 15-9, 16-1
 - maintaining, 12-8
 - processing, 12-3
 - processing for spatial predicates, 14-3
 - processing for XPath predicates, 13-4
 - rebuilding, 12-8, 12-9, 15-4
 - storing, 12-8
 - tuning, 12-2, 12-5
 - tuning for XPath predicates, 13-5
 - usage, 12-7
- indexing, 12-1
 - and database objects, 12-8
 - predicates, 12-1
 - spatial predicates, 14-3
 - XPath predicates, 13-2
- INSERT_EXPRESSION privilege, 11-11
- INSERT privilege, 11-11
- 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, 16-1

M

- matching rules
 - conflict resolution, 3-3
 - order of rule execution, 3-4
- metadata
 - expression set, 11-3
- multitier mode

rule evaluation, 6-4

O

object types

AnyData.convertObject method, 11-10, 15-2

attribute set, C-1

event structure, 2-1

Expression Filter

EXF\$ATTRIBUTE, 17-1

EXF\$ATTRIBUTE_LIST, 17-1

EXF\$INDEXOPER, 17-1

EXF\$TABLE_ALIAS, 17-1

EXF\$XPATH_TAG, 17-1

EXF\$XPATH_TAGS, 17-1

Rules Manager

RLM\$EVENTIDS, 7-1

objects created for the rule class

specifying storage properties, 3-9

order of rule execution, 3-4

P

predicate operators, 12-2

predicate table, 12-2, 12-8

querying, 12-5

predicates

evaluating in a sparse predicate group, 12-5

evaluating in a stored predicate group, 12-5

evaluating in an indexed predicate group, 12-5

indexable, 12-1

indexed, 12-4

sparse, 12-3, 12-5

spatial, 4-3, 14-1

stored, 12-4

XPath, 13-1

primitive event, 2-1, 2-5, 2-13, 6-1

as an XML document, 4-1

defined as XML document, 4-2

primitive events using relational tables

storing in relational tables, 4-1

privileges

granting, 11-11

revoking, 11-11

R

results view, 2-3

REVOKE_PRIVILEGE procedure, 11-11

RLM\$EVENTIDS object type, 7-1

rule, 2-2

defined, 1-2

rule action

preferences, 2-2

rule class, 2-2

process rules defined on XML documents, 4-1, 4-2

rule class properties

see event management policies

rule condition, 2-2, 5-1

ANY n semantics, 5-10

negation, 5-6

sequencing of primitive events, 5-5

set semantics, 5-10

with spatial predicates, 4-3

rule session, 2-3, 6-3

rules application

complex

XML tag extensions, 5-1

converting, D-3

creating

spanning multiple tiers, 2-13, 6-1

use composite event, 2-7

use simple or non-composite event, 2-5

event management policies, 2-12, 3-1

examples

law enforcement, 10-1

order management, 10-8

rule evaluation

multitier mode, 6-4

single tier mode, 6-4

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, 10-1

rules session, 6-1

S

secondary objects, 12-8

SELECT privileges, 11-11

single tier mode

rule evaluation, 6-4

sparse predicates, 12-5

spatial predicates, 4-3, 14-1

indexable, 14-3

using in expressions, 14-2

SQL*Loader, 16-1

stored predicates, 12-4

system triggers, C-2

T

table alias, A-1

attributes, 3-11, 4-1

tables

exporting, 16-2

importing, 16-2

triggers, C-2

system, C-2

validation, C-2

U

UNASSIGN_ATTRIBUTE_SET procedure, 11-8, 16-1

UPDATE EXPRESSION privilege, 11-11

UPDATE privilege, 11-11

- USER_EXPFIL_ASET_FUNCTIONS view, 19-1
- USER_EXPFIL_ATTRIBUTE_SETS view, 19-2
- USER_EXPFIL_ATTRIBUTES view, 19-2
- USER_EXPFIL_DEF_INDEX_PARAMS view, 19-2
- USER_EXPFIL_EXPRESSION_SETS view, 19-3
- USER_EXPFIL_EXPRSET_STATS view, 19-3
- USER_EXPFIL_INDEX_PARAMS view, 19-4
- USER_EXPFIL_INDEXES view, 19-4
- USER_EXPFIL_PREDTAB_ATTRIBUTES view, 19-5
- USER_EXPFIL_PRIVILEGES view, 19-5
- USER_EXPFIL_TEXT_INDEX_ERRORS view, 19-6
- USER_RLMGR_COMPRCLS_PROPERTIES view, 9-3
- USER_RLMGR_COMPRSET_PROPERTIES view, 9-3
- USER_RLMGR_EVENT_STRUCTS view, 9-1
- USER_RLMGR_PRIVILEGES view, 9-2
- USER_RLMGR_RULE_CLASS_STATUS view, 9-2
- USER_RLMGR_RULE_CLASSES view, 9-1
- users
 - exporting, 16-3

V

- VALIDATE_EXPRESSIONS procedure, 11-11, 16-2
- validation semantics, 11-10
- validation trigger, C-2

X

- XML schema
 - rule class properties, F-1
 - rule condition, F-3
- XML tags
 - extensions, 5-3
- XMLType datatype, 13-1
- XPath predicates, 13-1
 - configuring Expression Filter for, 13-4
 - index tuning for, 13-5
 - indexable, 13-3
 - indexing set of, 13-2