Oracle9*i* OLAP

Developer's Guide to the OLAP API

Release 2 (9.2)

March 2002

Part No. A95297-01

ORACLE

Oracle9*i* OLAP Developer’s Guide to the OLAP API, Release 2 (9.2)

Part No. A95297-01

Copyright © 2000, 2002, Oracle Corporation. All rights reserved.

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

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

If the Programs are delivered to the U.S. Government or anyone licensing or using the programs on behalf of the U.S. Government, the following notice is applicable:

Restricted Rights Notice Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

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

Oracle is a registered trademark, and Oracle9*i* is a trademark or registered trademark of Oracle Corporation. Other names may be trademarks of their respective owners.

Contents

Send Us Your Comments xi

Preface xiii

Audience xiv
Organization xiv
Related Documentation xv
Conventions xvii
Documentation Accessibility xxi

1 Introduction to the OLAP API

OLAP API Overview 1-2
Multidimensional Concepts And the OLAP API 1-2
What Type Of Data Can an Application Access Through the OLAP API? 1-3
What Can an Application Do with the OLAP API? 1-4
Context for OLAP API development 1-4
Access to Data and Metadata Through the OLAP API 1-5
MDM Model in the OLAP API 1-5
Access to Data Through the OLAP API 1-6
User Connection Requirements 1-7
OLAP API Client Software 1-7
Software Configurations 1-8
Requirements for Using the OLAP API Client Software 1-8

Developing an OLAP API Application 1-8
Step 1: Decide on General Design Issues 1-9
Step 2: Decide on Requirements for End-User Queries 1-9
Step 3: Design OLAP API Template Objects That Create End-User Queries 1-10
Step 4: Write and Test the Java Code for the Application 1-11
Step 5: Deploy the Application to users 1-12
Tasks That an OLAP API Application Performs 1-12
Task 1: Connect to the Data Store 1-12
Task 2: Discover the Available Metadata 1-13
Task 3: Select and Calculate Data Through Queries 1-13
Task 4: Retrieve Query Results 1-14

2 Understanding OLAP API Metadata

Overview of the OLAP API Metadata 2-2
Data Preparation 2-2
Metadata Preparation 2-2
OLAP Metadata Objects 2-2
Dimensions in the OLAP Metadata 2-3
Measures in the OLAP Metadata 2-3
Measure Folders in the OLAP Metadata 2-4
Overview of MDM Metadata Objects in the OLAP API 2-5
Mapping of OLAP Metadata Objects to MDM objects 2-6
MdmSchema Class 2-6
MdmSource Class 2-7
MdmDimension Class 2-8
Description of an MdmDimension 2-8
Information Held by an MdmDimensionDefinition 2-9
Information Held by an MdmDimensionMemberType 2-10
MdmLevel Class 2-10
Description of an MdmLevel 2-10
Elements of an MdmLevel 2-11

MdmHierarchy Class ... 2-12
Description of an MdmHierarchy ... 2-12
Elements of a Level MdmHierarchy ... 2-13
Level MdmHierarchy for Calendar Year ... 2-13
Level MdmHierarchy for Fiscal Year ... 2-15
Terminology: Nodes and leaves ... 2-16
Elements of a union MdmHierarchy ... 2-16
Distinct elements in the regions of a union MdmHierarchy ... 2-16
Union MdmHierarchy for Time ... 2-17
MdmListDimension Class ... 2-18
Description of an MdmListDimension ... 2-18
Elements of an MdmListDimension ... 2-18
MdmMeasure Class ... 2-19
Description of an MdmMeasure ... 2-19
Elements of an MdmMeasure ... 2-20
MdmMeasure Elements Are Determined by MdmDimension Elements ... 2-20
MdmMeasure with two MdmDimension objects ... 2-21
MdmAttribute Class ... 2-23
Description of an MdmAttribute ... 2-23
Elements of an MdmAttribute ... 2-23
Data Type and Type of MDM Metadata Objects ... 2-24
Data Type of MDM Metadata Objects ... 2-24
Getting the Data Type of an MdmSource ... 2-26
Type of MDM Metadata Objects ... 2-28
Getting the Type of an MdmSource ... 2-29

3 Connecting to a Data Store

Overview of the Connection Process ... 3-2
Connection Steps ... 3-2
Prerequisites for Connecting ... 3-2
Establishing a Connection ... 3-2
Step 1: Load the JDBC Driver ... 3-3
Step 2: Get a Connection from the DriverManager ... 3-3
Step 3: Create a TransactionProvider ... 3-4
Step 4: Create a DataProvider ... 3-4

Getting an Existing Connection 3-4
Executing DML Commands Through the Connection 3-5
Closing a Connection 3-5

4 Discovering the Available Metadata

Overview of the Procedure for Discovering Metadata 4-2
MDM Metadata 4-2
Purpose of Discovering the Metadata 4-2
Steps in Discovering the Metadata 4-3
Discovering Metadata and Making Queries 4-3
Creating an MdmMetadataProvider 4-3
Getting the Root MdmSchema 4-4
Function of the Root MdmSchema 4-4
Calling the getRootSchema Method 4-6
Getting the Contents of the Root MdmSchema 4-6
Getting the MdmDimension Objects in an MdmSchema 4-6
Getting the Subschemas in an MdmSchema 4-6
Getting the Contents of Subschemas 4-6
Getting the Measure MdmDimension and Its Contents 4-6
Getting the Characteristics of Metadata Objects 4-7
Getting the MdmDimension Objects for an MdmMeasure 4-7
Getting the Related Objects for an MdmDimension 4-7
Getting the Source for a Metadata Object 4-8
Sample Code for Discovering Metadata 4-9
Code for the SampleMetadataDiscoverer Program 4-10
Output from the SampleMetadataDiscoverer Program 4-16

5 Introduction to Querying

Characteristics of Source Objects 5-2
Source Type 5-2
Source Structure: Inputs and Outputs 5-3

Creating Source Objects 5-5
Getting Source Objects From Metadata Objects 5-5
Creating a Source from MdmDimension, MdmHierarchy, or MdmLevel Objects 5-6
Creating a Source from MdmMeasure or MdmAttribute Objects 5-7
Creating New Source Objects Using Source Methods 5-7
Creating Simple Nondimensional Source Objects 5-8
Creating Source Objects that Represent OLAP API Data Types 5-9

6 Making Queries Using Source Methods

Selecting Based on Source Value 6-2
Selecting Based on Output Values 6-3
Using the join Method to Change Inputs to Outputs 6-3
Effect of Input-Output Order on Source Structure 6-3
Changing Inputs to Outputs with timesDim as the First Output Created 6-4
Changing Inputs to Outputs with productsDim as the First Output Created 6-5
Selecting Based on Output Values and Source Values: Example 6-6
Selecting Values Based on Rank 6-6
Finding the Position of Values 6-6
Finding the Positions of Values When There are no Inputs or Outputs 6-8
Finding the Positions of Values When There Are Outputs and Inputs 6-8
Values Ranked in Ascending or Descending Order 6-9
Values Ranked in the Same or the Opposite Order as the Values of Another Source 6-9
Minimum Ranking 6-10
Maximum Ranking 6-10
Average Ranking 6-11
Packed Ranking 6-11
Percentile Ranking 6-11
nTile Ranking 6-12
Selecting Values Based on Hierarchical Position 6-12
Creating a Primary Source that Represents a Default Hierarchy 6-13
Creating a Primary Source for the Parent-Child Relationship 6-14
Creating Source Objects for Other Relationships 6-14
Drilling Down a Hierarchy: Example 6-15
Creating a Source that is a Self-Relation 6-16

Performing Numerical Analysis 6-18
Performing Numerical Operations 6-19
Subtracting the Same Value From all Values: Example 6-20
Subtracting the Values of one NumberSource from Another: Example 6-21
Making Numerical Comparisons 6-22
Working with Standard Numerical Functions 6-23
Working with Aggregation Methods 6-24
Calculating the Sum When a Source Has only Outputs: Example 6-25
Calculating the Sum When a Source Has an Output and an Input: Example 6-26
Creating Your own Numerical Functions 6-27
Creating Your own Standard Function: Example 6-27
Creating Your own Aggregation Function: Example 6-29
Manipulating String Values 6-29

7 Using a TransactionProvider

About Creating a Query in a Transaction 7-2
Types of Transaction Objects 7-3
Preparing and Committing a Transaction 7-3
About Transaction and Template Objects 7-5
Beginning a Child Transaction 7-5
About Rolling Back a Transaction 7-7
Getting and Setting the Current Transaction 7-8
Using TransactionProvider Objects 7-8

8 Understanding Cursor Classes and Concepts

Overview of the OLAP API Cursor Objects 8-2
Sources For Which You Cannot Create a Cursor 8-3
Cursor Objects and Transaction Objects 8-4
Cursor Class 8-4
Structure of a Cursor 8-5
Specifying the Behavior of a Cursor 8-8

CursorManagerSpecification Class 8-9
CursorSpecification Class 8-10
CursorInput Class 8-11
CursorManager Class 8-12
Updating the CursorManagerSpecification for a CursorManager 8-12
CursorManager Class Hierarchy 8-13
CursorManagerUpdateListener Class 8-15
CursorManagerUpdateEvent Class 8-15
About Cursor Positions and Extent 8-16
Positions of a ValueCursor 8-16
Positions of a CompoundCursor 8-17
About the Parent Starting and Ending Positions in a Cursor 8-22
What is the Extent of a Cursor? 8-25
About Fetch Sizes and Fetch Blocks 8-27
About Determining the Shape of a Fetch Block 8-29
About Sharing Fetch Blocks 8-29

9 Retrieving Query Results

Retrieving the Results of a Query 9-2
Getting Values from a Cursor 9-3
Navigating a CompoundCursor for Different Displays of Data 9-10
Specifying the Behavior of a Cursor 9-19
Calculating Extent and Starting and Ending Positions of a Value 9-21
Specifying Fetch Sizes and Fetch Blocks 9-24

10 Creating Dynamic Queries

About Template Objects 10-2
About Creating a Dynamic Source 10-2
About Translating User Interface Elements into OLAP API Objects 10-3
Overview of Template and Related Classes 10-3
What Is the Relationship Between the Classes That Produce a Dynamic Source? 10-4
Template Class 10-6
MetadataState Interface 10-6
SourceGenerator Interface 10-6
DynamicDefinition Class 10-7

Designing and Implementing a Template 10-7
Implementing the Classes for a Template 10-9
Implementing an Application That Uses Templates 10-14

A Setting Up the Development Environment

Overview A-2
Required Software A-2
Setting Up on Your Application Development Computer A-3
Installing the jar files A-3
Installing the OLAP API Javadoc A-3
Using a Sample Program A-3
Considerations for Deploying Your Application A-4

Index

Send Us Your Comments

Oracle9*i* OLAP Developer's Guide to the OLAP API, Release 2 (9.2)

Part No. A95297-01

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

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

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

- Electronic mail: `infodev_us@oracle.com`
- FAX: 781-238-9850 Attn: Oracle OLAP
- Postal service:
  Oracle Corporation
  Oracle OLAP Documentation
  10 Van de Graff Drive
  Burlington, MA 01803
  U.S.A.

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

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

Preface

The *Oracle9i OLAP Developer's Guide to the OLAP API* introduces Java programmers to the Oracle OLAP API which is the Java application programming interface for Oracle OLAP. Through Oracle OLAP, the OLAP API provides access to data stored in an Oracle database. The OLAP API's capabilities for querying, manipulating, and presenting data are particularly suited to applications that perform Online Analytical Processing.

This preface contains these topics:

- Audience
- Organization
- Related Documentation
- Conventions
- Documentation Accessibility

Audience

Oracle9i OLAP Developer's Guide to the OLAP API is intended for Java programmers who are responsible for creating applications that perform analysis using Oracle OLAP.

To use this document, you need be familiar with Java, relational database management systems, data warehousing, and Oracle OLAP and Online Analytical Processing (OLAP) concepts.

Organization

This document contains:

Chapter 1, "Introduction to the OLAP API"

Introduces the OLAP API to application developers who plan to use it in their Java applications.

Chapter 2, "Understanding OLAP API Metadata"

Describes the metadata objects that the OLAP API provides, and explains how these objects relate to the metadata objects that a database administrator specifies when preparing the data using the OLAP Metadata APIs.

Chapter 3, "Connecting to a Data Store"

Explains the procedure for connecting to a data store through the OLAP API.

Chapter 4, "Discovering the Available Metadata"

Explains the procedure for discovering the metadata in a data store through the OLAP API.

Chapter 5, "Introduction to Querying"

Introduces `Source` objects which are the OLAP API objects that are the specifications for sets of data that you use when making queries.

Chapter 6, "Making Queries Using Source Methods"

Discusses how to make queries using `Source` methods.

Chapter 7, "Using a TransactionProvider"

Describes the Oracle OLAP API `Transaction` and `TransactionProvider` interfaces and describes how you use implementations of those interfaces in an application. You must create a `TransactionProvider` before you can create a `DataProvider`, and you must use methods on the `TransactionProvider` to prepare and commit a `Transaction` before you can create a `Cursor` for a derived `Source`.

Chapter 8, "Understanding Cursor Classes and Concepts"

Describes the Oracle OLAP API `Cursor` class and its related classes, which you use to retrieve and gain access to the results of a query. This chapter also describes the `Cursor` concepts of position, fetch size, and extent.

Chapter 9, "Retrieving Query Results"

Describes how to retrieve the results of a query with an Oracle OLAP API `Cursor`, how to gain access to those results, and how to customize the behavior of a `Cursor` to fit your method of displaying the results.

Chapter 10, "Creating Dynamic Queries"

Describes the Oracle OLAP API `Template` class and its related classes, which you use to create dynamic queries. This chapter also provides examples of implementations of those classes.

Appendix A, "Setting Up the Development Environment"

Describes the steps you take to set up your development environment for creating applications that use the OLAP API.

Related Documentation

For more information, see these Oracle resources:

- Oracle 9*i* OLAP API Javadoc—Provides reference information for the Java packages that are the Oracle OLAP API.
- *Oracle9i OLAP User's Guide* — Describes how to use Oracle OLAP. It introduces the basic concepts underlying business analysis and multidimensional querying, as well as the basic tools used for application development and system administration.
- *Oracle9i OLAP Developer's Guide to the OLAP DML* — Explains how application developers can perform complex data analysis tasks (such as forecasts, models,

allocations, and some types of non-additive aggregation) by using the OLAP DML.

- *Oracle9i JDBC Developer's Guide and Reference*—Provides task-oriented and reference information about Oracle's Java Database Connectivity (JDBC) product that provides the basis for accessing data from Java programs, as well as Oracle-specific extensions to this Java standard.
- *Oracle9i Data Warehousing Guide* — Discusses the database structures, concepts, and issues involved in creating a data warehouse to support OLAP solutions.

Many of the examples in this book use the sample schemas of the seed database, which is installed by default when you install Oracle. Refer to *Oracle9i Sample Schemas* for information on how these schemas were created and how you can use them yourself.

In North America, printed documentation is available for sale in the Oracle Store at

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

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

`http://www.oraclebookshop.com/`

Other customers can contact their Oracle representative to purchase printed documentation.

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

`http://otn.oracle.com/admin/account/membership.html`

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

`http://otn.oracle.com/docs/index.htm`

To access the database documentation search engine directly, please visit

`http://tahiti.oracle.com`

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples
- Conventions for Windows Operating Systems

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface denotes book titles and emphasis.	*Oracle9i OLAP User's Guide* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`Bold`	Bold font denotes terms being defined for the first time,	The methods of the `Source` class and its subclasses return new `Source` objects sometimes called **derived** `Source` objects.
`UPPERCASE monospace (fixed-width) font`	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	The return value from its `getHierarchyType` method is LEVEL_HIERARCHY.
`lowercase monospace (fixed-width) font`	Lowercase monospace typeface indicates denotes Java program names, file names, path names, and Internet addresses.	Back up the datafiles and control files in the `/disk1/oracle/dbs` directory.

Convention	Meaning	Example
`MixedCase monospace (fixed-width) font`	Mixedcase monospace typeface is used for names of classes and interfaces and for multi-word names of variables, methods, and packages. The names of classes and interfaces begin with an upper-case letter. In all multi-word names, the second and succeeding words also begin with an upper-case letter.	To obtain access to the metadata, an application uses the `getRootSchema` method in `MdmMetadataProvider`.
`lowercase italic monospace (fixed-width) font`	Lowercase italic monospace font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate Java, SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
Source unitCost = mdmUnitCost.getSource;
```

The following table describes typographic conventions used in Java code examples and provides examples of their use.

Convention	Meaning
{ }	Braces enclose a block of statements.
//	A double slash begins a single-line comment, which extends to the end of a line.
/* */	A slash-asterisk and an asterisk-slash delimit a multi-line comment, which can span multiple lines/
. . .	Horizontal ellipsis shows that statements or clauses irrelevant to the discussion were left out.

Conventions for Windows Operating Systems

The following table describes conventions for Windows operating systems and provides examples of their use.

Convention	Meaning	Example
Choose Start >	How to start a program.	To start the Database Configuration Assistant, choose Start > Programs > Oracle - *HOME_NAME* > Configuration and Migration Tools > Database Configuration Assistant.
File and directory names	File and directory names are not case sensitive. The following special characters are not allowed: left angle bracket (<), right angle bracket (>), colon (:), double quotation marks ("), slash (/), pipe (|), and dash (-). The special character backslash (\) is treated as an element separator, even when it appears in quotes. If the file name begins with \\, then Windows assumes it uses the Universal Naming Convention.	`c:\winnt"\"system32 is the same as` `C:\WINNT\SYSTEM32`
`C:\>`	Represents the Windows command prompt of the current hard disk drive. The escape character in a command prompt is the caret (^). Your prompt reflects the subdirectory in which you are working. Referred to as the *command prompt* in this manual.	`C:\oracle\oradata>`
Special characters	The backslash (\) special character is sometimes required as an escape character for the double quotation mark (") special character at the Windows command prompt. Parentheses and the single quotation mark (') do not require an escape character. Refer to your Windows operating system documentation for more information on escape and special characters.	`C:\>exp scott/tiger TABLES=emp QUERY=\"WHERE job='SALESMAN' and sal<1600\"` `C:\>imp SYSTEM/`*`password`*` FROMUSER=scott TABLES=(emp, dept)`

Convention	Meaning	Example
HOME_NAME	Represents the Oracle home name. The home name can be up to 16 alphanumeric characters. The only special character allowed in the home name is the underscore.	`C:\> net start Oracle`*HOME_NAME*`TNSListener`
ORACLE_HOME and *ORACLE_BASE*	In releases prior to Oracle8*i* release 8.1.3, when you installed Oracle components, all subdirectories were located under a top level *ORACLE_HOME* directory that by default used one of the following names: ■ `C:\orant` for Windows NT ■ `C:\orawin98` for Windows 98 This release complies with Optimal Flexible Architecture (OFA) guidelines. All subdirectories are not under a top level *ORACLE_HOME* directory. There is a top level directory called *ORACLE_BASE* that by default is `C:\oracle`. If you install the latest Oracle9*i* release on a computer with no other Oracle software installed, then the default setting for the first Oracle home directory is `C:\oracle\orann` where `nn` is the latest release number. The Oracle home directory is located directly under *ORACLE_BASE*. All directory path examples in this guide follow OFA conventions. Refer to *Oracle9i Database Getting Started for Windows* for additional information about OFA compliances and for information about installing Oracle products in non-OFA compliant directories.	Go to the *ORACLE_BASE*`\`*ORACLE_HOME*`\rdbms\admin` directory.

Documentation Accessibility

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

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

Accessibility of Code Examples in Documentation JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

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

1 Introduction to the OLAP API

This chapter introduces the Oracle OLAP API to application developers who plan to use it in their Java applications.

This chapter includes the following topics:

- OLAP API Overview
- Access to Data and Metadata Through the OLAP API
- OLAP API Client Software
- Developing an OLAP API Application
- Tasks That an OLAP API Application Performs

OLAP API Overview

The OLAP API is a Java application programming interface (API) through which an application can access data for online analytical processing (OLAP). It is the API that is supplied with Oracle OLAP, an Oracle component.

The purpose of the OLAP API is to facilitate the development of OLAP applications, which allow users to dynamically select, aggregate, calculate, and perform other analytical tasks on data through a graphical user interface. Typically, the user interface of an OLAP application displays data in multidimensional formats, such as graphs and crosstabs.

In general, OLAP applications are developed within the context of business intelligence and data warehousing systems, and the features of the OLAP API are optimized for this type of application. With the OLAP API, a Java application can access, manipulate, and display data in multidimensional terms. The OLAP API also makes it possible to define a query in a step-by-step process that allows for undoing individual query steps without recreating the entire query. Such multistep queries are easy to modify and refine dynamically.

Multidimensional Concepts And the OLAP API

Data warehousing and OLAP applications are based on a multidimensional view of data, and they work with queries that represent selections of data. The following definitions introduce concepts that reflect the multidimensional view and are basic to data warehousing, OLAP, and the OLAP API:

- Dimension. A structure that categorizes data. Commonly used dimensions are customer, product, and time. Typically, a dimension is associated with one or more hierarchies. Several distinct dimensions, combined with measures, enable end users to answer business questions. For example, a Time dimension that categorizes data by month helps to answer the question, "Did we sell more widgets in January or June?"
- Measure. Data, usually numeric and additive, that can be examined and analyzed. Typically, a given measure is categorized by one or more dimensions, and it is described as “dimensioned by” them.
- Hierarchy. A logical structure that uses ordered levels as a means of organizing dimension elements in parent-child relationships. Typically, end users can expand or collapse the hierarchy by drilling down or up on its levels.
- Level. A position in a hierarchy. For example, a time dimension might have a hierarchy that represents data at the day, month, quarter, and year levels.

- Attribute. A descriptive characteristic of the elements of a dimension that an end user can specify to select data. For example, end users might choose products using a Color attribute.
- Query. A specification for a particular set of data, which is referred to as the query's result set. The specification may require selecting, aggregating, calculating, or otherwise manipulating data. If such manipulation is required, it is an intrinsic part of the query.

Two additional data warehouse and OLAP concepts, cube and edge, are not intrinsic to the OLAP API, but are often incorporated into the design of applications that use the OLAP API.

- Cube. A logical organization of multidimensional data. Typically, the edges of a cube contain dimension values, and the body of a cube contains measure values. For example, sales data can be organized into a cube whose edges contain values from the time, product, and customer dimensions and whose body contains values from the sales measure.
- Edge. One side of a cube. Each edge contains values from one or more dimensions. Although there is no limit to the number of edges on a cube, data is often organized for display purposes along three edges, which are referred to as the row edge, column edge, and page edge.

For more information about all of these concepts, see the *Oracle Data Warehousing Guide*.

What Type Of Data Can an Application Access Through the OLAP API?

The OLAP API, as part of Oracle OLAP, makes it possible for Java applications (including applets) to access data that resides in an Oracle data warehouse. A data warehouse is a relational database that is designed for query and analysis, rather than transaction processing. Warehouse data often conforms to a star schema, which represents a multidimensional data model. The star schema consists of one or more fact tables and one or more dimension tables that are related through foreign keys. Typically, a data warehouse is created from a transaction processing database by an extraction transformation transport (ETT) tool, such as Oracle Warehouse Builder.

In order for the OLAP API to access the data in a given data warehouse, a database administrator must first ensure that the data warehouse is configured according to an organization that is supported by Oracle OLAP. The star schema is one such organization, but not the only one. Once the data is organized in the warehouse, the database administrator must use the OLAP Metadata APIs to create the required metadata, which can be defined as "data about the data." Finally, with the metadata

in place, an application can access both the data and the metadata through the OLAP API.

See the *Oracle9i OLAP User's Guide* for information about supported data warehouse configurations and about using the OLAP Metadata APIs.

The collection of warehouse data for which a database administrator has created metadata using the OLAP metadata API is referred to as the data store to which the OLAP API gives access. Of course, each user who accesses data through the OLAP API might have security restrictions that limit the scope of the data that he or she can access within the data store.

What Can an Application Do with the OLAP API?

Through the OLAP API, an application can do the following:

- Establish a connection to a data store.
- Explore the metadata to discover what data is available for viewing or analysis.
- Create queries that manipulate the data according to the needs of application users (for example, selecting, aggregating, and calculating data).
- Retrieve query results that are structured for display in multidimensional format.
- Modify existing queries, rather than totally redefine them, as application users refine their analyses.

Context for OLAP API development

The OLAP API is a Java API, so it has all the advantages of the Java environment. It is platform independent, and it provides the benefits of an object-oriented API, such as abstraction, encapsulation, polymorphism, and inheritance. These strengths are built into the OLAP API, and because the client application is written in Java, its code can also take advantage of them.

In order to work with the OLAP API, application developers should have familiarity with Java, object-oriented programming, relational databases, data warehousing, and multidimensional OLAP concepts.

Access to Data and Metadata Through the OLAP API

OLAP API metadata describes the data that is available to the OLAP API through a given connection. The metadata records three things:

- The fact that a given set of data exists. For example, a sales measure exists in the data store.
- The structure of that set of data. For example, the sales measure is dimensioned by customer, product, and time.
- The characteristics of that set of data. For example, the sales measure contains numeric values, and it has a descriptive name that can be used in reports.

In contrast, the fact that 3542 dollars worth of boys outerwear was sold in Atlanta during January 1999 is data, not metadata.

These examples distinguish between the metadata and the data for a measure called Sales. The OLAP API makes a similar distinction between the metadata and the data for dimensions. For example, the fact that a product dimension exists and that it has text values as elements is metadata. In contrast, the fact that one of its elements is "boys outerwear" is data.

MDM Model in the OLAP API

The OLAP API's multidimensional metadata (MDM) model describes data in multidimensional terms, which are familiar to OLAP and data warehousing audiences. For example, it includes objects for measures, dimensions, hierarchies, and attributes.

The following are some of the Java classes that are supplied by the OLAP API in its implementation of the MDM model:

- `MdmMeasure`
- `MdmDimension`
- `MdmHierarchy`
- `MdmLevel`
- `MdmAttribute`
- `MdmSchema`
- `MdmMetadataProvider`

An MdmSchema is a container for MdmMeasure, MdmDimension, and other MdmSchema objects. An MdmSchema corresponds to a measure folder in the OLAP management feature of Oracle Enterprise Manager. Note that an MdmSchema does not necessarily correspond to a relational schema.

An MdmMetadataProvider gives an application access to metadata objects that were created by a database administrator using the OLAP management feature of Oracle Enterprise Manager. To obtain access to the metadata, an application uses the getRootSchema method in MdmMetadataProvider. This method returns the top-level MdmSchema, which contains all the MdmMeasure and MdmDimension objects that are accessible through this particular MdmMetadataProvider. The MdmDimension and MdmMeasure objects might be organized in a hierarchical tree, with subschemas nested under the top-level schema. Using the getMeasures, getDimensions, and getSubSchemas methods on all the nested MdmSchema objects, an application navigates through the metadata and discovers what data is available. In addition, the application can use methods to obtain the related MdmHierarchy, MdmLevel, and MdmAttribute objects.

Chapter 2, "Understanding OLAP API Metadata" provides detailed information about the OLAP API metadata.

Access to Data Through the OLAP API

An MdmMeasure or MdmDimension represents data in the data store. For example, an MdmMeasure called salesAmount might represent a set of numeric elements whose values are dollar sales figures, and an MdmDimension called productDim might represent a set of text elements whose values are product names. However, an application cannot create a query on the data using an MdmMeasure or MdmDimension. As metadata, MdmMeasure and MdmDimension objects provide descriptive information about data, but they do not provide the ability to query on that data. And an application must create a query in order to select, calculate, and otherwise manipulate data for analysis.

In order to create a query on the data for an MdmMeasure or MdmDimension, an application calls the getSource method on the MdmMeasure or MdmDimension. This method creates a Source object that represents the data for the purpose of querying. A Source is a specification for a query that defines a result set, and in this case, the result set is the data for the MdmMeasure or MdmDimension.

In addition to representing the data for metadata objects, Source objects can represent the data for any query that an application creates. For example, a Source might specify a query for a selection of MdmDimension values (January, February, March) or a calculation of the values of one MdmMeasure minus those of another

(`salesAmount` minus `unitCost`). An application can use the powerful methods on `Source` and its subclasses to combine data in any way that the user requires. And each new query is a new `Source`.

When an application prepares to display the data for a given `Source`, it creates a `Cursor` for the `Source`. The application then uses this `Cursor` to request and retrieve the data from the OLAP service. When an application makes a request for data, it can specify the typical amount of data that it requires at a given time (for example, enough to fill a 40-cell table on the screen). The OLAP service then handles the issues related to efficient retrieval. The application does not need to manage the timing, sizing, and caching of the data blocks that it retrieves through the OLAP API.

Because the primary focus of most OLAP applications is making queries against the data store, a significant proportion of their data manipulation code works with the following classes, each of which has methods for selecting, calculating, and otherwise manipulating data.

- `Source`
- `BooleanSource`
- `NumberSource`
- `StringSource`

One of the useful characteristic of `Source` objects is that they make no distinction between dimensions and measures. All `Source` objects behave in the same way.

User Connection Requirements

In addition to ensuring that data and metadata have been prepared appropriately, an application developer must ensure that application users can make a connection to the data store through the OLAP API and that users have database privileges that give them access to the data. For information about setting up for such connections, see the *Oracle9i OLAP User's Guide.*

OLAP API Client Software

The OLAP API client software is a set of Java packages containing classes that implement the programming interface to Oracle OLAP. An application calls the methods on these classes for discovering, querying, processing, and retrieving data.

When a Java application calls methods on OLAP API Java classes, it uses the OLAP API client software to communicate with Oracle OLAP, which resides within an

Oracle database instance. The communication between the OLAP API client software and Oracle OLAP is provided through Java Database Connectivity (JDBC), which is a standard Java interface for connecting to relational databases. For more information about JDBC, see the *Oracle9i JDBC Developer's Guide and Reference.*

Software Configurations

An application that uses the OLAP API client software (that is, calls methods in OLAP API classes) can reside on a single computer, or it can be divided into separate parts on two different computers. For example, the end-user portion can be separate from the portion that makes OLAP API calls. In this case, software on three computers could be involved.

For information about possible configurations, see the *Oracle9i OLAP User's Guide.*

Requirements for Using the OLAP API Client Software

To use the OLAP API classes as you develop your application, import them into your Java code in the standard way. When you deliver your application to users, include the OLAP API classes with the application. You must also ensure that users can access JDBC.

In order to develop an OLAP API application, you must have the Java Development Kit (JDK) from Sun Microsystems. Users must have a Java Runtime Environment (JRE) whose version number is compatible with the JDK you used for development.

For information about Java version requirements and about setting up the OLAP API client software, see Appendix A, "Setting Up the Development Environment". For detailed information about the OLAP API classes and methods, see the OLAP API Javadoc and subsequent chapters of this guide.

Developing an OLAP API Application

As an application developer, you perform the following steps to create an OLAP API application:

1. Decide on general design issues.
2. Decide on requirements for end-user queries.
3. Design OLAP API `Template` objects that create end-user queries. This is an optional step.
4. Write and test the Java code for the application.

5. Deploy the application to users.

The rest of this topic presents a general description of each step.

Step 1: Decide on General Design Issues

Consider broad questions such as the following:

- Will the application be a standalone application (two-tier architecture), or will it be divided, with end-user code on a separate tier from the data manipulation code (three-tier architecture)?
- Will the application always access the same known metadata (for example, describing employee data whose structure is constant), or must it discover what metadata is available every time it makes a connection?

Step 2: Decide on Requirements for End-User Queries

Specify, in as much detail as possible, the nature of the queries that the end user will be able to make. Because the OLAP API makes it possible to define queries in a step-by-step process, it is also important to decide on the query modification capabilities that the application will offer the user. Consider questions such as the following:

- By what criteria will the end user select data through the application's dialog boxes? For example, will the application present a list of dimensions? Can the user drill up and down on the hierarchy of a dimension? Are there attributes of dimensions that the user can specify for selecting data (for example, color or size)? Can the user make selections based on data values (for example, population over 20,000)?
- As the user refines a query through a series of steps, can the user undo a step in the process to return the query to an earlier state?
- As the user refines a query, can the user specify the scope of an undo request? For example, the undo request might apply only to the values of one field out of many in the selection dialog box.

Planning the end-user queries is a crucial step in the application design process, so you should complete it as thoroughly as possible. Ideally, you should create an end-user query model that identifies all the conceptual query objects with which the application user interface will deal. This strategy takes advantage of the strengths of object-oriented design, and it allows for a clear correspondence between user interface objects and OLAP API objects.

The following are examples of conceptual query objects for an application user interface:

- Dimension. This object has hierarchies on which the user can drill and attributes from which the user can select.
- Dimension selection. This object represents a selection of dimension elements.
- Edge. This object represents one side of a cube and has related dimension objects.
- Cube. This multidimensional object has related edge objects. It also has a related measure.

Each of these conceptual query objects can be represented by an OLAP API `Template` object.

Step 3: Design OLAP API Template Objects That Create End-User Queries

An optional step in implementing an OLAP API application is designing `Template` objects. This step is recommended because, the use of `Template` objects offers the following benefits:

- Dynamic queries. With a `Template`, you can create a modifiable query. That is, when you have created one query and you want to execute another one that is similar but not identical, you do not have to create an entirely new query. You simply make a small change to the existing query. Thus, the query is dynamic, rather than static.
- Refinement and rollback of queries. With a `Template`, you can capture a series of steps that a user has completed when specifying a query. Each step refines the query further and is recorded as a new query state. If the user decides to cancel one or more of the specification steps, you can rollback the query to an earlier state.
- Matching of code to user interface characteristics. When you design a `Template`, you can make it correspond directly to the operations that a user performs. For example, if your application includes a balance sheet, you can create a balance sheet `Template` that incorporates all the appropriate characteristics (such as a method of aggregation) and behaviors (such as automatic totalling).

For a more detailed example of how Template objects mirror the query-building aspects of an application's user interface, imagine an application that allows the user to create a three-dimensioned cube of data through the following steps:

1. Choose a measure whose data will be in the cube.
2. Select the values for each dimension that will provide structure to the cube.
3. Specify the placement of the dimensions on the three edges of the cube.

As the application developer for this interface, you would design a Template subclass for each of the following objects: dimension, dimension selection, edge, and cube. As part of the design, you would specify methods on the Template subclasses that allow you to combine objects as needed. For example, the edge Template class might have an addDimension method, and the cube Template class might have an addEdge method. Once you have implemented the dimension, dimension selection, edge, and cube Template classes, you can use them again and again in your application. They are basic building blocks in your application's code for querying and manipulating data.

In this stage of the application design process, you should make detailed specifications for each Template in the application. For information about designing Template objects, see Chapter 10, "Creating Dynamic Queries".

Step 4: Write and Test the Java Code for the Application

Up to this step, you have not written any Java code. You have considered questions about the design of your application, and you have made detailed specifications for the Template objects that your application will include. Now you must do the following to implement the application:

1. Set up the OLAP API client software on your development computer, as described in Appendix A, "Setting Up the Development Environment". If you are designing a three-tiered application, the development computer (from the OLAP API point of view) is the middle-tier computer.
2. Identify the data store that you will use for developing and testing the application. Ensure that the data is structured as a star or snowflake schema in an Oracle data warehouse, and ensure that the OLAP management feature in Oracle Enterprise Manager has provided the metadata.
3. Write the Java classes for your application, importing the OLAP API classes as needed. Among the Java classes that you write, include the Template classes that you designed.
4. Test your application using the test data store.

For information about coding an application that uses the OLAP API, see the subsequent chapters of this guide and the OLAP API Javadoc. See "Tasks That an OLAP API Application Performs" on page 1-12 for a description of the tasks that an application typically performs.

Step 5: Deploy the Application to users

Keep the following in mind when you deploy your application:

- Include the OLAP API Java classes along with the ones that you have developed.
- Ensure that the user's computer (or the middle tier computer) has access to an Oracle database instance that includes the OLAP option.
- Ensure that the user has access to an appropriate Oracle data warehouse with metadata prepared by the OLAP Metadata APIs.
- Provide documentation for your application, giving installation instructions and explaining the user interface that you have created.

Tasks That an OLAP API Application Performs

An application that uses the OLAP API typically performs the following tasks:

1. Connect to the data store
2. Discover the available metadata
3. Select and calculate data through queries
4. Retrieve query results

The rest of this topic briefly describes these tasks, and the rest of this guide provides detailed information.

Task 1: Connect to the Data Store

An application connects to the data store by identifying some information about the target Oracle database and specifying this information in a JDBC connection method.

For more information about connecting, see Chapter 3, "Connecting to a Data Store".

Task 2: Discover the Available Metadata

Having established a connection, the application creates an `MdmMetadataProvider`. This object gives access to all the metadata objects in the data store.

To discover the available metadata, an application uses the `getRootSchema` method on the `MdmMetdataProvider` to obtain the top-level measure folder for all of its metadata objects. The application then gets the dimensions, measures, and subfolders that are under the root. Once the application has all the dimensions and measures, it can interrogate them to get their attributes, hierarchies, levels, and other characteristics.

Having determined the metadata objects that it has to work with, the application can present relevant lists of objects to the user for data selection and manipulation.

For a description of the metadata objects, see Chapter 2, "Understanding OLAP API Metadata". For information about how an application can discover the available metadata, see Chapter 4, "Discovering the Available Metadata".

Task 3: Select and Calculate Data Through Queries

The heart of any OLAP application lies in the construction of queries against the data store. The application user interface provides ways for the user to select data and specify what should be done with it. Then, the data manipulation code translates these instructions into queries against the data store. The queries can be as simple as a selection of dimension elements, or they can be complex, including several aggregations and calculations on measure values.

The OLAP API object that specifies a query is a `Source`. Therefore, a significant portion of any OLAP API application is devoted to dealing with `Source` objects.

You can manipulate `Source` objects directly, using methods such as `select`, `remove`, and `appendValues` to create selections. In addition, you can use methods such as `plus`, `div`, and `total` to calculate values. `Source` and its subclasses, `NumberSource`, `StringSource`, and `BooleanSource`, have a rich assortment of methods for manipulating data. The most powerful method in `Source` is `join`, which gives you the ability to combine `Source` objects in almost any way imaginable.

If you are implementing a simple user interface, you might use only the methods on the `Source` classes to select and manipulate the data that users specify in the interface. However, if you want to offer your users multistep selection procedures and the ability to modify queries or undo individual steps in their selections, you should use `Template` classes as described in the topic "Developing an OLAP API

Application" on page 1-8. Within the code for each Template, you use the methods on the Source classes, but the Template classes themselves allow you to modify and refine even the most complex query. In addition, you can minimize your work by writing general-purpose Template classes and reusing them in various parts of your application.

For information about working with Source objects, see Chapter 5, "Introduction to Querying". For information about working with Template objects, see Chapter 10, "Creating Dynamic Queries".

Task 4: Retrieve Query Results

When users of an OLAP application are selecting, calculating, combining, and generally manipulating data, they also want to see the results of their work. This means that the application must retrieve the result sets of queries from the data store and display the data in multidimensional form. To retrieve a result set for a query through the OLAP API, the application creates a Cursor based on the Source that specifies the query.

Because the OLAP API was designed to deal with a multidimensional view of data, a Source can have a multidimensional result set. For example, a Source can represent an MdmMeasure that is structured by three MdmDimension objects. The Cursor for this Source has a structure that mirrors the Source itself; that is, the Cursor organization is based on the same three MdmDimension objects.

To retrieve all the items of data through a Cursor, the application can loop through the multidimensional Cursor structure. This design is well adapted to the requirements of standard user interface objects for painting the computer screen. It is especially well adapted to the display of data in multidimensional format.

For more information about using Cursor objects to retrieve data, see Chapter 8, "Understanding Cursor Classes and Concepts".

2

Understanding OLAP API Metadata

This chapter describes the metadata objects that the OLAP API provides, and explains how these objects relate to the OLAP metadata objects that a database administrator specifies using the OLAP Metadata APIs.

This chapter includes the following topics:

- Overview of the OLAP API Metadata
- OLAP Metadata Objects
- Overview of MDM Metadata Objects in the OLAP API
- MdmDimension Class
- MdmLevel Class
- MdmHierarchy Class
- MdmListDimension Class
- MdmMeasure Class
- MdmAttribute Class
- Data Type and Type of MDM Metadata Objects

Overview of the OLAP API Metadata

The OLAP API provides a Java application with access to a multidimensional view of data in an Oracle database. The OLAP API design includes objects that are consistent with that view and are familiar to data warehousing and OLAP developers. For example, it has objects for measures, dimensions, hierarchies, levels, and attributes. The OLAP API design incorporates an object-oriented model called MDM (multidimensional metadata).

The data in an Oracle database must be prepared by a database administrator in order to support the MDM model. Even though recent SQL enhancements have introduced some multidimensional objects, such as dimension, there are other objects and characteristics that must be added.

Data Preparation

A database administrator starts with a data warehouse that is organized according to certain specifications. For example, it might conform to a star schema. The requirements are described in the *Oracle9i OLAP User's Guide*.

Metadata Preparation

Using the OLAP Metadata APIs, the administrator adds OLAP metadata to the data warehouse. The OLAP metadata objects, which are created in this step, supply the metadata required for Oracle OLAP to access the data. These OLAP metadata objects map to MDM metadata objects in the OLAP API.

The topic "OLAP Metadata Objects" on page 2-2 briefly describes the OLAP metadata objects that a database administrator prepares for use with Oracle OLAP.

OLAP Metadata Objects

Using the OLAP Metadata APIs, a database administrator adds OLAP metadata to a data warehouse. The end result is the creation of one or more measure folders that contain one or more measures. The measures have dimensions, and the dimensions have hierarchies, levels, and attributes. Each of these OLAP metadata objects maps directly to an MDM object in the OLAP API.

For detailed information about OLAP metadata and about using the OLAP Metadata APIs, see the *Oracle9i OLAP User's Guide*.

Note that the OLAP metadata includes a cube object, which does not map directly to any MDM object. Database administrators reference cubes in the OLAP Metadata

APIs when they specify the dimensions of each measure. Once the dimensions are specified, they are firmly associated with their measures in the metadata, so this type of cube object is not needed in the MDM model.

The rest of this topic briefly describes the OLAP metadata objects that map directly to MDM objects in the OLAP API.

Dimensions in the OLAP Metadata

The he following are some of the characteristics that a database administrator can specify for dimensions:

- General characteristics, such as the name of the dimension and the schema from which its data is drawn.
- Levels, which record the levels of the dimension. The database administrator typically specifies one or more levels for each OLAP dimension.
- Hierarchies, which specify the parent-child relationships between the levels. The database administrator typically specifies at least one hierarchy for each OLAP dimension. If there is only one level for the dimension, then no hierarchy is specified and the dimension is a simple, non-hierarchical list.
- Attributes, which record characteristics of the level elements for the dimension. For example, an attribute might record the gender of each customer in the customers dimension.

Typically, a database administrator specifies one or more columns in a database table to serve as the basis for each OLAP level, hierarchy, and attribute.

A database administrator creates cubes after creating dimensions. A cube is a set of dimensions that provide organizational structure for measures.

Measures in the OLAP Metadata

The OLAP Metadata APIs give a database administrator the ability to specify that a given measure belongs to a given cube. Because a cube is a set of dimensions that provide organizational structure for measures, specifying that a given measure belongs to a given cube specifies the dimensions of that measure. This is essential information for the OLAP API, where the dimensionality of a measure is one of its most important features.

To identify the data for a measure, the database administrator typically specifies a column in a fact table where the measure's data resides. As an alternative, the database administrator can specify a calculation or transformation that produces the data.

Measure Folders in the OLAP Metadata

Once a database administrator has created measures (first creating dimensions and cubes), the next step is to create one or more groups of measures called measure folders. Typically, the measures in a given folder are related by subject matter. That is, they all pertain to the same business area. For example, there might be three separate folders for financials, sales, and human resources.

The measures in a given measure folder can belong to different cubes, and they can be from more than one schema.

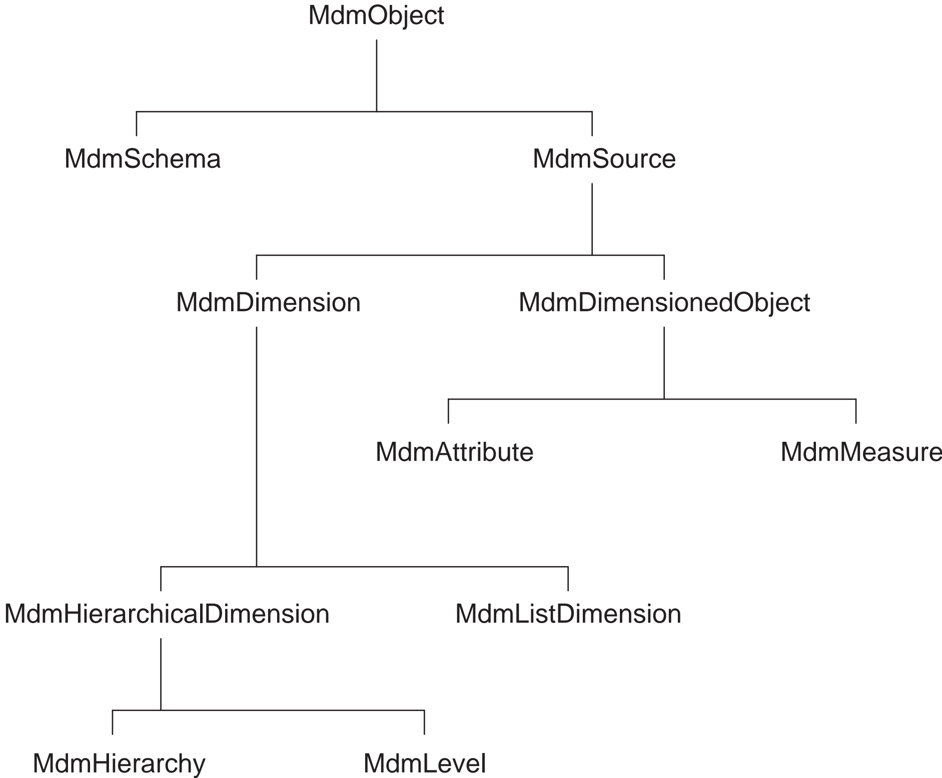
The database administrator must create at least one measure folder because the scope of the data that an OLAP API application can access is defined in terms of measure folders. That is, an OLAP API `MdmMetadataProvider` gives access only to the measures that are contained in measure folders. Of course, each measure's dimensions are included, along with its hierarchies, levels, and attributes.

In this context, it is important to understand that measure folders can be nested. This means that a given measure folder can have subfolders that have their own measures, and even their own subfolders. Thus, a database administrator can arrange measures in a hierarchy of folders, and an OLAP API `MdmMetadataProvider` can give access to all of the measure folders and their subfolders.

Overview of MDM Metadata Objects in the OLAP API

The OLAP API implementation of the MDM model is represented by classes in the `oracle.express.mdm` package. Most of the classes in this package implement metadata objects, such as dimensions and measures. The following diagram introduces the subclasses of the `MdmObject` class.

Figure 2–1 MdmObject Class and Its Subclasses.



Mapping of OLAP Metadata Objects to MDM objects

An application accesses metadata objects by creating an OLAP API MdmMetadataProvider and using it to discover the available metadata objects in the data store.

The metadata objects that a database administrator specifies using the OLAP Metadata APIs map directly to MDM metadata objects that are accessible through the MdmMetadataProvider. The following table presents the typical mapping.

OLAP Metadata Objects	MDM Metadata Objects
Dimension	MdmHierarchy or MdmListDimension
Hierarchy	MdmHierarchy
Level	MdmLevel
Measure	MdmMeasure
Attribute	MdmAttribute
Measure Folder	MdmSchema

This chapter describes the MDM metadata objects. For information about how an application discovers the available MDM metadata objects in the data store, see Chapter 4, "Discovering the Available Metadata".

MdmSchema and MdmSource are the two subclasses of MdmObject.

MdmSchema Class

An MdmSchema represents a set of data that is used for navigational purposes. An MdmSchema is a container for MdmMeasure, MdmDimension, and other MdmSchema objects. An MdmSchema is equivalent to a folder or directory that contains associated items. It does not correspond to a relational schema in the Oracle database. Instead, it corresponds to a measure folder, which can include data from several relational schemas and which was created by a database administrator using the OLAP Metadata APIs.

Data that is accessible through the OLAP API is arranged under a top-level MdmSchema, which is referred to as the root MdmSchema. Under the root, there are one or more subschemas. To begin navigating the metadata, an application calls the getRootSchema method on the MdmMetadataProvider, as explained in Chapter 2, "Understanding OLAP API Metadata".

The root MdmSchema contains all the MdmDimension objects that are in the data store. Most MdmDimension objects are contained in subschemas under the root MdmSchema. However, a data store can contain a dimension that is not included in a subschema. The root MdmSchema contains MdmDimension objects that are in subschemas as well as MdmDimension objects that are not.

The root MdmSchema contains MdmMeasure objects only if they are not contained in a subschema. Because most MdmMeasure objects belong to a subschema, the root MdmSchema typically has no MdmMeasure objects.

An MdmSchema has methods for getting all the MdmMeasure, MdmDimension, and MdmSchema objects that it contains. The root MdmSchema also has a method for getting the measure MdmDimension, whose elements are all the MdmMeasure objects in the data store regardless of whether they belong to a subschema.

MdmSource Class

An MdmSource represents a measure, dimension, or other set of data (such as an attribute) that is used for analysis. This abstract class is the basis for some important MDM metadata classes, such as MdmMeasure, MdmDimension, and MdmAttribute.

MdmSource objects represent data, but they do not provide the ability to create queries on that data. Their function is informational, recording the existence, structure, and characteristics of the data. They do not give access to the data values.

In order to access the data values for a given MdmSource, an application calls the getSource method on the MdmSource. This method returns a Source through which an application can create queries on the data represented by the MdmSource. The following line of code creates a Source from an MdmDimension called mdmProductsDim.

```
Source productsDim = mdmProductsDim.getSource();
```

A Source that is the result of the getSource method on an MdmSource is called a primary Source. An application creates new Source objects from this primary Source as it selects, calculates, and otherwise manipulates the data. Each new Source specifies a new query.

For more information about working with Source objects, see Chapter 5, "Introduction to Querying".

The rest of this chapter describes the subclasses of MdmSource, along with other classes, such as MdmDimensionDefinition and MdmDimensionMemberType, that are closely related.

MdmDimension Class

MdmDimension is a subclass of MdmSource.

Description of an MdmDimension

An MdmDimension represents a list of elements that can organize a set of data. For example, if you have a set of sales figures for a given year and you organize them by month, the list of months is a dimension of the sales data. The values of the month dimension act as indexes for identifying each particular value in the set of sales data.

In the OLAP API, the abstract MdmDimension class represents the general concept of a list of elements that can organize data. MdmDimension has an abstract subclass called MdmHierarchicalDimension, which represents a list that has hierarchical characteristics.

The following concrete subclasses of MdmDimension represent the specific kinds of MdmDimension objects that can be used in analysis.:

- MdmLevel, which represents a list of elements that supply one level of a hierarchical structure. Each element can have a parent and one or more children. The parents and children of a given MdmLevel element are not within the given MdmLevel. They are elements of different MdmLevel objects.
- MdmHierarchy, which represents a list of elements arranged in a hierarchical structure that has levels based on parent-child relationships. Each element can have a parent and one or more children, and all of these elements are within the MdmHierarchy.

  Though the parent and child elements are within the MdmHierarchy, they correspond to elements in MdmLevel objects. Therefore, loosely speaking, an MdmHierarchy is composed of MdmLevel objects. Some MdmHierarchy objects are simply composed of MdmLevel objects. Others are unions of one or more subordinate MdmHierarchy objects, which in turn, are composed of MdmLevel objects.
- MdmListDimension, which represents a simple list of elements that play no part in any hierarchical structure. The elements have no parents and no children.

Both MdmLevel and MdmHierarchy are concrete subclasses of the abstract MdmHierarchicalDimension class.

An MdmDimension can have one or more MdmAttribute objects. Each of these objects maps the elements of the MdmDimension to values representing some characteristic of the elements. To obtain the MdmAttribute objects for a given MdmDimension, call its getAttributes method.

An MdmDimension has an MdmDimensionDefinition, which represents the structure of the underlying data, and an MdmDimensionMemberType, which represents the basic nature of the elements. These two objects hold important information about the MdmDimension to which they belong. For a given MdmDimension, you use its getDefinition and getMemberType methods to obtain these related objects.

Information Held by an MdmDimensionDefinition

An MdmDimensionDefinition indicates the structure of the underlying data on which the MdmDimension is based. The MdmDimensionDefinition class is abstract. Therefore, instances are always one of the following subclasses:

- MdmBaseDimensionDefinition, which indicates that the MdmDimension has underlying data structured as a single list. For example, an MdmLevel is often based on a single column in a relational table.
- MdmUnionDimensionDefinition, which indicates that the MdmDimension has underlying data structured as the union of two or more lists. For example, an MdmHierarchy can be based on two or more columns in a relational table, one column for each MdmLevel.
- MdmAliasDimensionDefinition, which indicates that the MdmDimension acts as a proxy (that is, an alias) for another MdmDimension.

An MdmDimension that has an MdmUnionDimensionDefinition has regions. A region of a given MdmDimension is another MdmDimension that represents a subset of the elements of the given MdmDimension. For example, an MdmDimension for calendar year might have one region that represents quarters and another region that represents months. To obtain the regions of an MdmDimension, you call the getRegions method on its MdmUnionDimensionDefinition.

Information Held by an MdmDimensionMemberType

An `MdmDimensionMemberType` indicates the basic nature of the elements in the `MdmDimension`. It holds a description for each element, and it often provides methods for finding out other information about individual elements. The `MdmDimensionMemberType` class is abstract. Therefore, instances are always one of the following subclasses:

- `MdmTimeMemberType`, which indicates that the `MdmDimension` elements represent time periods. An `MdmTimeMemberType` has methods for finding out the end date and time span for each element.
- `MdmMeasureMemberType`, which indicates that the `MdmDimension` elements are all the `MdmMeasure` objects in the data store. There is only one `MdmDimension` with an `MdmMeasureMemberType`, and it is referred to as the measure `MdmDimension`. You can obtain the measure `MdmDimension` by calling the `getMeasureDimension` method on the root `MdmSchema`.
- `MdmStandardMemberType`, which indicates that the `MdmDimension` elements have no specific characteristics. Most `MdmDimension` objects have an `MdmStandardMemberType`.

MdmLevel Class

`MdmLevel` is a subclass of `MdmHierarchicalDimension`, which is an abstract subclass of `MdmDimension`.

Description of an MdmLevel

An `MdmLevel` is an `MdmHierarchicalDimension` whose parents and children are elements from other `MdmLevel` objects. The elements from a given `MdmLevel` correspond to a subset of the elements in an `MdmHierarchy`.

A given `MdmLevel` is based on a level that was specified by a database administrator using the OLAP Metadata APIs. Typically, the database administrator specified a column in a database table to provide the elements for the level.

Even though the elements of an `MdmLevel` have parent-child relationships, an `MdmLevel` is represented as a simple list. The parent-child relationships among the elements are recorded in the parent and ancestors attributes, which you can obtain by calling the `getParentRelation` and `getAncestorsRelation` methods on the `MdmLevel`. Sometimes the parent and ancestors attributes are referred to as parent and ancestors relations.

Typically, an MdmLevel has an MdmBaseDimensionDefinition, because the underlying data is structured as a single list.

Elements of an MdmLevel

The list of elements in an MdmLevel includes only the elements in that one level. The values of the elements must be unique. However, uniqueness can be achieved by a database administrator who defines the level using two relational columns. For example, a level that represents cities can be defined in the relational database based on both the city column and the state column. This makes it possible for the value “Springfield” to appear for two different elements in the city level: one appears for Springfield, Illinois and another appears for Springfield, Massachusetts.

The following table lists the elements for an MdmLevel called mdmQuarter, which records the three-month quarters for a level MdmHierarchy called mdmTimesDimCalHier. This MdmHierarchy covers four years, so the number of elements in mdmQuarter is 16.

Elements of mdmQuarter
1998-Q1
1998-Q2
1998-Q3
1998-Q4
1999-Q1
1999-Q2
1999-Q3
1999-Q4
2000-Q1
. . .
2001-Q4

MdmHierarchy Class

MdmHierarchy is a subclass of MdmHierarchicalDimension, which is an abstract subclass of MdmDimension.

Description of an MdmHierarchy

An MdmHierarchy is an MdmHierarchicalDimension that includes all the elements of one or more hierarchical structures. That is, all the parents and children are within the MdmHierarchy.

Even though the parent-child relationships exist in the MdmHierarchy, its elements are represented as a simple list. The relationships among the elements are recorded in the parent and ancestors attributes, which you can obtain by calling the getParentRelation and getAncestorsRelation methods on the MdmHierarchy. You can obtain the region for each element by calling the getRegionAttribute method on the MdmDimensionDefinition of the MdmHierarchy. Sometimes the parent, ancestors, and region attributes are referred to as parent, ancestors, and region relations.

Typically, an MdmHierarchy is one of the following types:

- Level MdmHierarchy, which represents a hierarchical structure whose regions are MdmLevel objects. For example, a level MdmHierarchy for calendar year might have as its regions MdmLevel objects for year, quarter, month and day.

  A level MdmHierarchy has an MdmUnionDimensionDefinition, and its regions are MdmLevel objects. The return value from its getHierarchyType method is LEVEL_HIERARCHY. A level MdmHierarchy is based on a hierarchy that was defined by a database administrator using the OLAP Metadata APIs.

- Union MdmHierarchy, which represents a dimension that has one or more subordinate hierarchical structures. These structures are represented by one or more level or value MdmHierarchy objects. An example, of an MdmHierarchy with two structures is a union MdmHierarchy for time that has two regions, one for the calendar year and another for the fiscal year. Each region is a level MdmHierarchy.

  A union MdmHierarchy has an MdmUnionDimensionDefinition and its regions are MdmHierarchy objects. The return value from its getHierarchyType method is UNION_HIERARCHY. A union

MdmHierarchy is based on a dimension that was defined as having one or more hierarchies by a database administrator using the OLAP Metadata APIs.

- Value MdmHierarchy, which represents a hierarchical structure whose elements have parents and children but no levels and therefore no regions. For example, a company's employee reporting structure can be represented with parent/child relationships but without levels.

  A value MdmHierarchy has an MdmBaseDimensionDefinition. The return value from its getHierarchyType method is VALUE_HIERARCHY. A value MdmHierarchy is based on a dimension that was flagged as a value hierarchy by a database administrator using the OLAP Metadata APIs.

When working with MdmHierarchy objects in the current release of the OLAP API, keep the following points in mind.

- Call the getAttributes method on a union MdmHierarchy, not on its subordinate level or value MdmHierarchy objects or on MdmLevel objects.
- Create queries on Source objects that are based on a level or value MdmHierarchy, not on a union MdmHierarchy.
- Call the getParentRelation and getAncestorsRelation methods on a level or value MdmHierarchy, not on a union MdmHierarchy.
- Call the getRegionAttribute method on the MdmUnionDimensionDefinition of a level MdmHierarchy, not of a union MdmHierarchy. This method returns the MdmAttribute that records the MdmLevel to which each MdmHierarchy element belongs.

Elements of a Level MdmHierarchy

The elements of a level MdmHierarchy include all of the elements of all of its regions. The values of the elements in a particular level MdmHierarchy must be unique. The following examples present the elements of two level MdmHierarchy objects, one for calendar year and the other for fiscal year.

Level MdmHierarchy for Calendar Year

The following table lists the values of the elements for a level MdmHierarchy called mdmTimesDimCalHier, which includes the elements from four MdmLevel objects: mdmYear, mdmQuarter, mdmMonth, and mdmDay. The number of elements

is 1529: 4 year elements, 16 quarter elements, 48 month elements, and 1461 day elements.

Elements of `mdmTimesDimCalHier`
1998
1998-Q1
1998-01
01-JAN-98
02-JAN-98
03-JAN-98
. . .
01-FEB-98
02-FEB-98
03-FEB-98
. . .
1998-Q2
1998-04
01-APR-98
02-APR-98
03-APR-98
. . .
1999
1999-Q1
1999-01
01-JAN-99
02-JAN-99
03-JAN-99
. . .

Level MdmHierarchy for Fiscal Year

The following table lists the values of the elements for a level `MdmHierarchy` called `mdmTimesDimFisHier`, which includes the elements from four `MdmLevel` objects: `mdmFisYear`, `mdmFisQuarter`, `mdmFisMonth`, and `mdmFisDay`. The number of elements is 1529: 4 fiscal year elements, 16 fiscal quarter elements, 48 fiscal month elements, and 1461 fiscal day elements.

In this example, the `mdmFisDay` `MdmLevel` is based on the same relational database column on which the `mdmDay` `MdmLevel` is based (see the earlier example for calendar year). Therefore, the values of the elements for these two `MdmLevel` objects are identical. However, this does not mean that the elements themselves are identical. The elements in `mdmDay` are distinct from the elements in `mdmFisDay`; only the values of the two sets of elements are the same.

Elements of `timesDimFisHier`
FIS-1998
FIS-1998-Q1
FIS-1998-01
01-JUL-98
02-JUL-98
03-JUL-98
. . .
01-AUG-98
02-AUG-98
03-AUG-98
. . .
FIS-1998-Q2
FIS-1998-04
01-OCT-98
02-OCT-98
03-OCT-98
. . .

Elements of `timesDimFisHier`
FIS-1999
FIS-1999-Q1
FIS-1999-01
01-JUL-99
02-JUL-99
03-JUL-99
. . .

Terminology: Nodes and leaves

A level `MdmHierarchy` represents a tree structure with parent-child relationships. Elements in the lowest `MdmLevel` are referred to as leaves, and the elements in the `MdmLevel` objects above the lowest level are referred to as nodes. Nodes have children; leaves do not.

Elements of a union MdmHierarchy

The elements of a union `MdmHierarchy` include all of the elements of all of its regions. Another way to say this is that a union `MdmHierarchy` includes all of the elements of all of the `MdmLevel` objects in all of its subordinate `MdmHierarchy` objects. In hierarchical terms, the set of elements includes all of the leaves (the elements at the lowest level) and all of the nodes (the elements at the levels above the lowest one) for all the hierarchies.

Distinct elements in the regions of a union MdmHierarchy

The elements in the regions of a union `MdmHierarchy` are totally distinct. That is, a given element does not appear in more than one region of a union `MdmHierarchy`. This is the case even if the database administrator specified the same level in two different hierarchies of a dimension. When this happens, Oracle OLAP creates two different `MdmLevel` objects, one for each level `MdmHierarchy`.

Though the elements of a union `MdmHierarchy` are distinct, the values of the elements are not required to be unique. Therefore in the example below, the leaf elements of the two regions of the union `MdmHierarchy` have values that are identical.

Union MdmHierarchy for Time

Consider a union MdmHierarchy called mdmTimesDim, which has two regions. The first region is the MdmHierarchy called mdmTimesDimCalHier, which has 1529 elements. The second region is the MdmHierarchy called mdmTimesDimFisHier, which also has 1529 elements. The set of elements for mdmTimesDim is the union of the elements from its two MdmHierarchy objects. Because no element can appear in both MdmHierarchy objects, mdmTimesDim has 3058 elements. Note that a calendar year begins on January 1, while a fiscal year begins on July 1.

The following table lists the values of the elements of the union MdmHierarchy called mdmTimesDim. To distinguish the elements of mdmDay and mdmFisDay, whose values are identical, the word "(fiscal)" appears next to the values for mdmFisDay. The mdmDay and mdmFisDay objects were introduced earlier in the examples for the elements of a level MdmHierarchy.

Elements of mdmTimesDim
1998
1998-Q1
1998-01
01-JAN-98
. . .
1999
1999-Q1
1999-01
01-JAN-99
. . .
FIS-1998
FIS-1998-Q1
FIS-1998-01
01-JUL-98 (fiscal)
. . .

Elements of mdmTimesDim
FIS-1999
FIS-1999-Q1
FIS-1999-01
01-JUL-99 (fiscal)
. . .

MdmListDimension Class

MdmListDimension is a subclass of MdmDimension.

Description of an MdmListDimension

An MdmListDimension is a simple list of elements that have no hierarchical characteristics. That is, the notion of having a parent or a child is not relevant for the elements of an MdmListDimension.

Elements of an MdmListDimension

A given MdmListDimension is based on a dimension that was specified as having a single level and no hierarchy by a database administrator using the OLAP Metadata APIs.

The following table lists the values of the elements of an MdmListDimension called mdmColor.

Elements of mdmColor
Black
Blue
Cyan
Green
Magenta
Red
Yellow
White

MdmMeasure Class

MdmMeasure is a subclass of MdmDimensionedObject, which is an abstract subclass of MdmSource.

Description of an MdmMeasure

An MdmMeasure represents a set of data that is organized by one or more MdmDimension objects. The structure of the data is similar to that of a multidimensional array. Like the dimensions of an array, the MdmDimension objects that organize an MdmMeasure provide the indexes for identifying individual cells.

For example, suppose you have an MdmMeasure for sales data, and the data is organized by product, time, customer, and channel (with channel representing the marketing method, such as direct or indirect.). You can think of the data as occupying a four-dimensional array with the product, time, customer and channel dimensions providing the organizational structure. The values of these four dimensions are indexes for identifying each particular cell in the array, which contains a single sales value. You must specify a value for each dimension in order to identify a value in the array. In relational terms, the MdmDimension objects constitute a compound (that is, composite) primary key for the MdmMeasure.

The values of an MdmMeasure are usually numeric, but this is not necessary.

Elements of an MdmMeasure

A given MdmMeasure is based on an OLAP measure that was created by a database administrator using the OLAP Metadata API. In most cases, the database administrator specified a column in a fact table to act as the basis for the OLAP measure (alternatively, the database administrator specified a mathematical calculation or a data transformation). In many but not all cases, the database administrator also specified at least one hierarchy for each of the measure's OLAP dimensions, as well as an aggregation method. Oracle OLAP uses all of this information to identify the number of elements in the MdmMeasure and the value of each element.

MdmMeasure Elements Are Determined by MdmDimension Elements

The set of elements that are in an MdmMeasure is determined by the structure of its MdmDimension objects. That is, each element of an MdmMeasure is identified by a unique combination of elements from its MdmDimension objects.

Typically, the MdmDimension objects of an MdmMeasure are union MdmHierarchy objects. That is, they have at least one hierarchical structure. It is important to remember that the elements of a union MdmHierarchy include all of the leaves and all of the nodes for all of the level MdmHierarchy objects that represent its regions. Because of this structure, the values of the elements of an MdmMeasure are of two kinds:

- Values from the fact table column (or fact-table calculation) on which the MdmMeasure is based, as specified using the OLAP Metadata APIs. These values belong to MdmMeasure elements that are identified by a combination of leaf MdmHierarchy elements.
- Aggregated values that Oracle OLAP has provided. These values belong to MdmMeasure elements that are identified by at least one node element from an MdmHierarchy.

As an example, imagine an MdmMeasure called mdmUnitCost that is dimensioned by union MdmHierarchy objects called mdmTimesDim and mdmProductsDim. Each MdmHierarchy has leaf elements (for example, 01-JAN-99 in mdmTimesDim), and each MdmHierarchy has node elements (for example, 1999-Q1 in mdmTimesDim). A unique combination of two elements, one from each MdmHierarchy, identifies each mdmUnitCost element, and every possible combination is used to specify the entire mdmUnitCost element set.

Some mdmUnitCost elements are identified by a combination of leaf elements (for example, a particular product item and a particular month). Other mdmUnitCost elements are identified by a combination of node elements (for example, a

particular product group and a particular quarter). Still other mdmUnitCost elements are identified by a mixture of leaf and node elements. The values of the mdmUnitCost elements that are identified only by leaf elements come directly from the column in the database fact table (or fact table calculation). They represent the lowest level of data. However, for the elements that are identified by at least one node element, Oracle OLAP provides the values. These higher-level values represent aggregated, or rolled-up data.

Thus, the data represented by an MdmMeasure is a mixture of fact table data from the data store and aggregated data that Oracle OLAP makes available for analytical manipulation.

MdmMeasure with two MdmDimension objects

The table below lists values for some of the elements of the MdmMeasure called mdmUnitCost, which is described above. This MdmMeasure has mdmProductsDim and mdmTimesDim as its MdmDimension objects. Each of these objects is a union MdmHierarchy with regions that are level MdmHierarchy objects. For example, the level MdmHierarchy objects for mdmTimesDim are mdmTimesDimCalHier and mdmTimesDimFisHier, and the level MdmHierarchy for mdmProductsDim is mdmProductsDimHier.

Because there are so many elements in the MdmMeasure, the table shows only a few of them. For example, for mdmTimesDim, you should imagine that the ellipses (indicated by dots) cover additional days, months, quarters, and years in the mdmTimesDimCalHier region, as well as the entire mdmTimesDimFisHier region.

mdmProductsDimHier has three levels, which represent the product category (such as Boys), the product subcategory (such as Outerwear - Boys), and the individual product item (such as #23110). The table shows only one element from each level, and the ellipses cover all the rest.

Almost all the elements shown in the table are aggregated. The ones that are *not* aggregated are marked with an asterisk. These nonaggregated elements are the ones

that are identified by the lowest level elements of both `mdmProductsDim` and `mdmTimesDim`.

Elements of **mdmProductsDim**	Elements of **mdmTimesDim**	Elements of **mdmUnitCost**
Boys	1998	12,800,444.00
Boys	1998-Q1	4,563,150.00
Boys	1998-01	1,837,254.00
Boys	01-JAN-98	185,346.00
Boys	02-JAN-98	232,590.00
Boys	03-JAN-98	155,403.00
. . .	. . .	. . .
Outerwear -Boys	1998	6,473,065.00
Outerwear -Boys	1998-Q1	2,000,317.00
Outerwear -Boys	1998-01	637,482.00
Outerwear -Boys	01-JAN-98	27,009.00
Outerwear -Boys	02-JAN-98	20,346.00
Outerwear -Boys	03-JAN-98	12,498.00
. . .	. . .	. . .
23110	1998	847,362.00
23110	1998-Q1	200,635.00
23110	1998-01	60,735.00
23110	01-JAN-98	2,226.00 *
23110	02-JAN-98	1,709.00 *
23110	03-JAN-98	2,047.00 *
. . .	. . .	. . .

MdmAttribute Class

MdmAttribute is a subclass of MdmDimensionedObject, which is an abstract subclass of MdmSource.

Description of an MdmAttribute

An MdmAttribute represents a particular characteristic of the elements of an MdmDimension. An MdmAttribute maps one element of the MdmDimension to a particular value. A typical example is an MdmAttribute that records the gender of each customer in an MdmDimension called mdmCustomersDim. In this case, the elements of the MdmAttribute have the values “Female” and “Male”.

The values of an MdmAttribute might be String values (such as “Female”), numeric values (such as 45), or objects (such as MdmLevel objects).

Like an MdmMeasure, an MdmAttribute has elements that are organized by its MdmDimension. For example, the gender MdmAttribute has one element (with “Female” or “Male” as its value) for each element of the MdmDimension called mdmCustomersDim.

Typically, not all of the elements of an MdmDimension have meaningful mappings to the values of a given MdmAttribute. For example, the gender MdmAttribute applies only to the lowest level of mdmCustomersDim, because gender makes no sense for higher levels such as cities or states. If an MdmAttribute does not apply to some elements of an MdmDimension, then their MdmAttribute values are null.

Some MdmAttribute objects provide a mapping that is one-to-many, rather than one-to-one. Therefore, a given element in an MdmDimension might map to a whole set of MdmAttribute elements. For example, the MdmAttribute that serves as the ancestors attribute for an MdmHierarchy maps each MdmHierarchy element to its set of ancestor MdmHierarchy elements.

Elements of an MdmAttribute

A given MdmAttribute is based on an attribute that was specified for a dimension or a level by a database administrator using the OLAP Metadata APIs.

The following table lists the elements for an MdmAttribute called mdmCustomersDimGender, which is based on the MdmDimension called mdmCustomersDim. Note that the values of the MdmAttribute are null for the

city, country, and region levels. There are meaningful values only for the customer level, where each customer is represented by a number.

Elements of `mdmCustomersDim`	Elements of `mdmCustomersDimGender`
...	...
Africa	`null`
South Africa	`null`
Cape Town	`null`
5420	Female
11650	Female
17880	Male
24120	Female
67720	Male
73960	Male
...	...

Data Type and Type of MDM Metadata Objects

All `MdmSource` objects have the following two basic characteristics:

- Data type
- Type

Data Type of MDM Metadata Objects

The concept of data type is a familiar one in computer languages and database technology. It is common to categorize data into types such as integer, Boolean, and string.

The OLAP API implements the concept of data type through the `FundamentalMetadataObject` and `FundamentalMetadataProvider` classes. Every data type recognized by the OLAP API is represented by a `FundamentalMetadataObject`, and you obtain this object by calling a method on a `FundamentalMetadataProvider`.

The following table lists the most familiar OLAP API data types. For each data type, the table presents a description of the `FundamentalMetadataObject` that

represents the data type and the name of the method in `FundamentalMetadataProvider` that returns the object.

OLAP API Data Type	Description of the FundamentalMetadataObject	Method in FundamentalMetadataProvider
Boolean	Represents the data type that corresponds to the Java `boolean` data type.	`getBooleanDataType`
Date	Represents the data type that corresponds to the Java `Date` class.	`getDateDataType`
Double	Represents the data type that corresponds to the Java `double` data type.	`getDoubleDataType`
Float	Represents the data type that corresponds to the Java `float` data type.	`getFloatDataType`
Integer	Represents the data type that corresponds to the Java `int` data type.	`getIntegerDataType`
Short	Represents the data type that corresponds to the Java `short` data type.	`getShortDataType`
String	Represents the data type that corresponds to the Java `String` class.	`getStringDataType`

In addition to these familiar data types, the OLAP API includes two generalized data types (which represent groups of the familiar data types) and two data types

that represent the absence of values. The following table lists these additional data types.

OLAP API Data Type	Description of the `FundamentalMetadataObject`	Method in `FundamentalMetadataProvider`
Number	Represents a general data type that includes any or all of the following OLAP API numeric data types: Double, Float, Integer, and Short.	`getNumberDataType`
Value	Represents a general data type that includes any or all of the OLAP API data types.	`getValueDataType`
Empty	Represents missing data, for example when an `MdmSource` has no elements at all defined for it.	`getEmptyDataType`
Void	Represents `null` data, for example when an `MdmSource` has a single element that has a `null` value.	`getVoidDataType`

When an MDM metadata object, such as an `MdmMeasure`, has a given data type, this means that each one of its elements conforms to that data type. If the data type is numeric, then the elements also conform to the generalized Number data type, as well as to the specific data type (Double, Float, Integer, or Short). The elements of any MDM metadata object conform to the Value data type, as well as to their more specific data type, such as Integer or String.

If the elements of an object represent a mixture of several numeric and non-numeric data types, then the data type is only Value. The object has no data type that is more specific than that.

The MDM metadata objects for which data type is relevant are `MdmSource` objects, such as `MdmMeasure`, `MdmHierarchy`, and `MdmLevel`. The typical data type of an `MdmMeasure` is one of the numeric data types; the typical data type of an `MdmHierarchy` or `MdmLevel` is String.

Getting the Data Type of an MdmSource

If you have obtained an `MdmSource` from the data store, and you want to find out the data type of its elements, you call its `getDataType` method. This method returns a `FundamentalMetadataObject`.

To find out which OLAP API data type is represented by the returned `FundamentalMetadataObject`, you compare it to the `FundamentalMetadataObject` for each OLAP API data type. That is, you compare it to the return value of each of the data type methods in `FundamentalMetadataProvider`.

The following sample method returns a constant that indicates the data type of the `MdmSource` that is passed in as a parameter. Note that this code creates a `FundamentalMetadataProvider` by calling a method on a `DataProvider` (`dp`). Getting a `DataProvider` is described in Chapter 4, "Discovering the Available Metadata". Also note that the constants referenced in this method are defined elsewhere in the class to which the method belongs. The constants are not supplied by the OLAP API.

Example 2–1 Getting the Data Type of an MdmSource

```
public int getDataType(MdmSource metaSource) {

   int theDataType = 0;
   FundamentalMetadataProvider fmp =
        dp.getFundamentalMetadataProvider();

   if (fmp.getBooleanDataType() == metaSource.getDataType())
       theDataType = BOOLEAN_TYPE;
    else if (fmp.getDateDataType() == metaSource.getDataType())
       theDataType = DATE_TYPE;
    else if (fmp.getDoubleDataType() == metaSource.getDataType())
       theDataType = DOUBLE_TYPE;
    else if (fmp.getFloatDataType() == metaSource.getDataType())
       theDataType = FLOAT_TYPE;
    else if (fmp.getIntegerDataType() == metaSource.getDataType())
       theDataType = INTEGER_TYPE;
    else if (fmp.getShortDataType() == metaSource.getDataType())
       theDataType = SHORT_TYPE;
    else if (fmp.getStringDataType() == metaSource.getDataType())
       theDataType = STRING_TYPE;
    else if (fmp.getNumberDataType() == metaSource.getDataType())
       theDataType = NUMBER_TYPE;
    else if (fmp.getValueDataType() == metaSource.getDataType())
       theDataType = VALUE_TYPE;

    return theDataType;
    }
```

Type of MDM Metadata Objects

An MDM metadata object, such as an `MdmSource`, is a collection of elements. Its type (as opposed to its data type) is another metadata object from which the given metadata object draws its elements. In other words, the elements of a given metadata object correspond to a subset of the elements in its type. There can be no element in the metadata object that does not match an element of its type.

Consider the following example of a union `MdmHierarchy` called `mdmCustomersDim`, which has the OLAP API data type of String. `mdmCustomersDim` has a region (a level `MdmHierarchy` called `mdmCustomersDimGeogHier`), which in turn has its own regions (`MdmLevel` objects). In each case, the region represents a subset of elements. In the following list, the regions are indented under the `MdmHierarchy` to which they belong.

```
mdmCustomersDim
      mdmCustomersDimGeogHier
            mdmGeogTotal
            mdmRegion
            mdmSubregion
            mdmCountry
            mdmState
            mdmCity
            mdmCustomer
```

Because of the hierarchical structure, `mdmCountry` (for example) draws its elements from the elements of `mdmCustomersDimGeogHier`. That is, the set of elements for `mdmCountry` corresponds to a subset of elements from `mdmCustomersDimGeogHier`, and `mdmCustomersDimGeogHier` is the type of `mdmCountry`.

Similarly, `mdmCustomersDimGeogHier` is a region of `mdmCustomersDim`. Therefore, `mdmCustomersDimGeogHier` draws its elements from `mdmCustomersDim`, which is its type.

However, `mdmCustomersDim` is not a region of any other object. It is the top of the hierarchy. The pool of elements from which `mdmCustomersDim` draws its elements is the entire set of possible String values. Therefore, the type of `mdmCustomersDim` is the `FundamentalMetadataObject` that represents the OLAP API String data type. In the case of `mdmCustomersDim`, the type and the data type are the same.

The following list presents the types that are typical for the most common `MdmSource` objects:

- The type of an `MdmLevel` is the level `MdmHierarchy` to which it belongs.
- The type of a level `MdmHierarchy` is the union `MdmHierarchy` to which it belongs.
- The type of a union `MdmHierarchy` is the `FundamentalMetadataObject` that represents its OLAP API data type. Typically, this is the String data type.
- The type of an `MdmMeasure` is the `FundamentalMetadataObject` that represents its OLAP API data type. Typically, this is one of the OLAP API numeric data types.

Getting the Type of an MdmSource

If you have obtained an `MdmSource` from the data store, and you want to find out its type, you call its `getType` method. This method returns the object that is the type of the `MdmSource` object.

For example, the following Java statement obtains the type of the `MdmLevel` called `mdmCountry`.

Example 2–2 Getting the Type of an MdmSource

```
MetadataObject mdmCountryType = ((MdmSource) mdmCountry).getType();
```

3

Connecting to a Data Store

This chapter explains the procedure for connecting to a data store through the OLAP API.

This chapter includes the following topics:

- Overview of the Connection Process
- Establishing a Connection
- Getting an Existing Connection
- Executing DML Commands Through the Connection
- Closing a Connection

Overview of the Connection Process

When an application accesses data through the OLAP API, it uses a connection provided by the Oracle implementation of Java Database Connectivity (JDBC) from Sun Microsystems. For information about using this JDBC implementation, see the *Oracle9i JDBC Developer's Guide and Reference.*

Connection Steps

The procedure for connecting involves loading an Oracle JDBC driver, getting a connection through that driver, and creating two OLAP API objects that handle transactions and data transfer.

These steps are described in the topic "Establishing a Connection" on page 3-2.

Prerequisites for Connecting

Before attempting to make an OLAP API connection to an Oracle database, ensure that the following requirements are met:

- The Oracle database instance is running and was installed with the OLAP option.
- Your Oracle database user ID has access to the relational schemas on which the data store is based.
- The Oracle client installation of the JDBC drivers is complete. For information about installing JDBC drivers, see the *Oracle9i JDBC Developer's Guide and Reference.*
- The OLAP API jar files are on the application development computer and are accessible to the application code. For information about setting up the OLAP API jar files, see Appendix A, "Setting Up the Development Environment"

Establishing a Connection

To make a connection, perform the following steps:

1. Load the JDBC driver that you will use.
2. Get a `Connection` from the `DriverManager`.
3. Create a `TransactionProvider`.
4. Create a `DataProvider`.

These steps are explained in more detail in the rest of this topic.

Note that the `TransactionProvider` and `DataProvider` objects that you create in these steps are the ones that you use throughout your work with the data store. For example, when you create certain `Source` objects, you use methods on this `DataProvider` object.

Step 1: Load the JDBC Driver

The following line of code loads a JDBC driver and registers it with the JDBC `DriverManager`.

Example 3–1 Loading the JDBC Driver for a Connection

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

After the driver is loaded, you can use the `DriverManager` object to make a connection. For more information about loading Oracle's JDBC drivers, see the *Oracle9i JDBC Developer's Guide and Reference.*

Step 2: Get a Connection from the DriverManager

The following code gets a JDBC `Connection` object from the `DriverManager`.

Example 3–2 Getting a JDBC Connection

```
String url = "jdbc:oracle:thin:@lab1:1521:orcl";
String user = "hepburn";
String password = "tracey";
oracle.jdbc.OracleConnection conn = (oracle.jdbc.OracleConnection)
   java.sql.DriverManager.getConnection(url, user, password);
```

This example connects user `hepburn` with password `tracey` to a database with SID (system identifier) `orcl`. The connection is made through TCP/IP listener port `1521` of host `lab1`. The connection uses the Oracle JDBC thin driver.

There are many ways to specify your connection characteristics using the `getConnection` method. See the *Oracle9i JDBC Developer's Guide and Reference* for details.

After you have the `Connection` object, you can create the required OLAP API objects, `TransactionProvider` and `DataProvider`.

Step 3: Create a TransactionProvider

`TransactionProvider` is an OLAP API interface. Therefore, in your code, you use an instance of the concrete class called `ExpressTransactionProvider`. The following line of code creates a `TransactionProvider`.

Example 3–3 Creating a TransactionProvider

```
ExpressTransactionProvider tp = new ExpressTransactionProvider();
```

A `TransactionProvider` is required for creating a `DataProvider`.

Step 4: Create a DataProvider

`DataProvider` is an OLAP API abstract class. Therefore, in your code, you use an instance of the concrete subclass called `ExpressDataProvider`. The following lines of code create and initialize a `DataProvider`.

Example 3–4 Creating a DataProvider

```
ExpressDataProvider dp = new ExpressDataProvider(conn, tp);
dp.initialize();
```

A `DataProvider` is required for creating a `MetadataProvider`, which is described in Chapter 4, "Discovering the Available Metadata"

Getting an Existing Connection

If you need access to the JDBC `Connection` object after the connection has been established, you can call the `getConnection` method on your `DataProvider`. The following line of code calls the `getConnection` method on a `DataProvider` called `dp`.

Example 3–5 Getting an Existing Connection

```
oracle.jdbc.OracleConnection currentConn = dp.getConnection();
```

Executing DML Commands Through the Connection

Some applications depend on the run-time execution of Oracle OLAP data manipulation language (DML) commands or programs. DML commands and programs execute in an analytic workspace outside the context of MDM metadata, which is intrinsic to the OLAP API. Therefore, such commands and programs do not operate on MDM objects, such as `MdmMeasure` and `MdmDimension`. Instead, they operate on DML objects, such as variable and dimension. The MDM and DML contexts are related but distinct.

To execute DML commands or programs in an analytic workspace, create an OLAP API `SLPExecutor` object, specifying the JDBC `Connection` object that you want to use. Note that the data manipulation language is sometimes referred to as a stored procedure language (SPL).

The following lines of code create and initialize an `SPLExecutor` object on a JDBC `Connection` object called `conn`.

Example 3–6 Executing DML Commands

```
SPLExecutor dmlExec = new SPLExecutor(conn);
dmlExec.initialize();
```

To specify an analytic workspace in which you want to execute DML commands, attach the workspace using the DML command called `AW`. For example, the following command executes the `AW` command for attaching a workspace named `mysales`.

```
string returnVal = dmlExec.execute('aw attach mysales');
```

For information about using the DML, see the *Oracle9i OLAP Developer's Guide to the OLAP DML* and the Oracle9i OLAP DML Reference help. For more information about using an `SPLExecutor`, see the OLAP API Javadoc.

Closing a Connection

When you have completed your work with the data store, use the `close` method on the JDBC `Connection` object. In the following sample code, the `Connection` object is called conn.

Example 3–7 Closing a Connection

```
conn.close();
```

If you are finished using the OLAP API, but you want to continue working in your JDBC connection to the database, use the `close` method on your `DataProvider` to release the OLAP API resources. In the following example code, the `DataProvider` is called `dp`.

```
dp.close();
```

4 Discovering the Available Metadata

This chapter explains the procedure for discovering the metadata in a data store through the OLAP API.

This chapter includes the following topics:

- Overview of the Procedure for Discovering Metadata
- Creating an MdmMetadataProvider
- Getting the Root MdmSchema
- Getting the Contents of the Root MdmSchema
- Getting the Characteristics of Metadata Objects
- Getting the Source for a Metadata Object
- Sample Code for Discovering Metadata

Overview of the Procedure for Discovering Metadata

The OLAP API provides access to a collection of Oracle data for which a database administrator has created OLAP metadata using the OLAP Metadata APIs. This collection of data is the data store for the application.

Potentially, the data store includes all of the measure folders that were created by the database administrator using the OLAP Metadata APIs. However, the scope of the data store that is visible when a given application is running depends on the database privileges that apply to the user ID through which the connection was made. A user sees all of the measure folders (as `MdmSchema` objects) that the database administrator created, but the user sees the measures and dimensions that are contained in those measure folders only if he or she has access rights to the relational tables on which the measures and dimensions are based.

MDM Metadata

When the database administrator created the metadata, the OLAP Metadata APIs created measures, dimensions, and other OLAP metadata objects. In the OLAP API, these objects are accessed as multidimensional metadata (MDM) objects, as described in Chapter 2, "Understanding OLAP API Metadata". The mapping between the OLAP metadata objects and the MDM objects is automatically performed by Oracle OLAP.

Purpose of Discovering the Metadata

The metadata objects in the data store help your application to make sense of the data. They provide a way for you to find out what data is available, how it is structured, and what its characteristics are.

Therefore, after connecting, your first step is to find out what metadata is available. Armed with this knowledge, you can present choices to the end user about what data should be selected or calculated and how it should be displayed.

Steps in Discovering the Metadata

Before investigating the metadata, your application must make a connection to Oracle OLAP, as described in Chapter 3, "Connecting to a Data Store". Then, your application performs the following steps:

1. Create an `MdmMetadataProvider`
2. Get the root `MdmSchema` from the `MdmMetadataProvider`
3. Get the contents of the root `MdmSchema`, which include `MdmMeasure`, `MdmDimension`, `MdmMeasureDimension`, and `MdmSchema` objects. In addition, get the contents of any subschemas.
4. Get the characteristics of each `MdmMeasure` and `MdmDimension`. For example, for each `MdmMeasure` get its `MdmDimension` objects, and for each `MdmDimension` find out whether it is a union `MdmHierarchy`, a level `MdmHierarchy`, an `MdmLevel`, or an `MdmListDimension`.

The next four topics in this chapter describe these steps in detail.

Discovering Metadata and Making Queries

After you discover the metadata, you typically go on to create queries for selecting, calculating, and otherwise manipulating the data. In order to work with data in these ways, you must get the `Source` objects that Oracle OLAP has created to represent the data for querying. These `Source` objects are referred to as primary `Source` objects.

This chapter focuses on the initial step of discovering the available metadata, but it also briefly mentions the step of getting a primary `Source` from a metadata object. Subsequent chapters of this guide explain how you work with primary `Source` objects and create queries based on them.

Creating an MdmMetadataProvider

An `MdmMetadataProvider` gives access to the metadata in a data store. It maps OLAP metadata objects, such as measures, dimensions, and measure folders, to the corresponding MDM objects, such as `MdmMeasure`, `MdmDimension`, and `MdmSchema`.

Before you can create an `MdmMetadataProvider`, you must create a `DataProvider` as described in Chapter 3, "Connecting to a Data Store".

The following code creates an MdmMetadataProvider using a DataProvider called dp.

Example 4–1 Creating an MdmMetadataProvider

```
MdmMetadataProvider mp = null;
mp = (MdmMetadataProvider) dp.getDefaultMetadataProvider();
```

Getting the Root MdmSchema

Getting the root MdmSchema is the first step in exploring the metadata in your data store.

Function of the Root MdmSchema

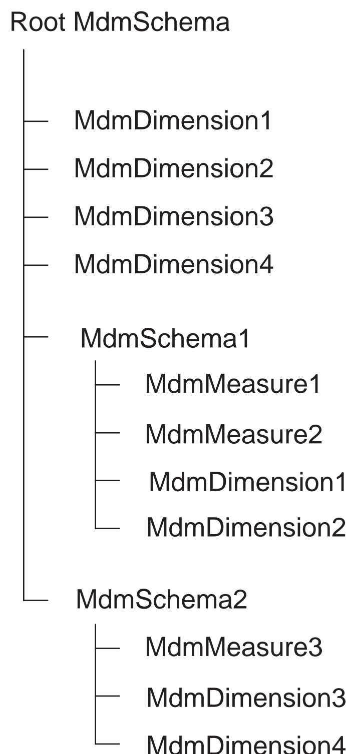
The metadata objects that are accessible through a given MdmMetadataProvider are organized in a tree-like structure, with the root MdmSchema at the top. Under the root MdmSchema are MdmDimension objects and one or more MdmSchema objects, which are referred to as subschemas. In addition, if there are any MdmMeasure objects that do not belong to a subschema, they are included under the root.

Subschemas have their own MdmMeasure and MdmDimension objects. Optionally, they can have their own subschemas as well.

The root MdmSchema contains all the MdmDimension objects that are in the subschemas. Therefore, a given MdmDimension typically appears twice in the tree. It appears once under the root MdmSchema and again under the subschema. If an MdmDimension does not belong to a subschema, it is listed only under the root.

The starting point for discovering the available metadata objects is the root MdmSchema, which is the top of the tree. The following diagram illustrates an MdmSchema that has two subschemas and four MdmDimension objects.

Figure 4–1 Root MdmSchema and Subschemas



Using the OLAP Metadata APIs, a database administrator arranges dimensions and measures under one or more top-level measure folders. When Oracle OLAP maps the measure folders to `MdmSchema` objects, it always creates the root `MdmSchema` above the `MdmSchema` objects for the top-level measure folders. Therefore, even if the database administrator creates only one measure folder, its corresponding `MdmSchema` will be a subschema under the root.

For more information about MDM metadata objects and how they map to OLAP metadata objects, see Chapter 2, "Understanding OLAP API Metadata".

Calling the getRootSchema Method

The following code gets the root MdmSchema for an MdmMetadataProvider called mp.

Example 4–2 Getting the Root MdmSchema

```
MdmSchema root = mp.getRootSchema();
```

Getting the Contents of the Root MdmSchema

The root MdmSchema contains MdmDimension objects, MdmSchema objects, and possibly MdmMeasure objects. In addition, the root MdmSchema has a measure MdmDimension that lists all the MdmMeasure objects.

Getting the MdmDimension Objects in an MdmSchema

The following code gets a List of MdmDimension objects that are in the MdmSchema called schema.

Example 4–3 Getting MdmDimension Objects

```
List dims = schema.getDimensions();
```

Getting the Subschemas in an MdmSchema

The following code gets a List of MdmSchema objects that are in the MdmSchema called schema.

Example 4–4 Getting Subschemas

```
List subSchemas = schema.getSubSchemas();
```

Getting the Contents of Subschemas

For each MdmSchema that is under the root MdmSchema, you can call the getMeasures, getDimensions, and getSubSchemas methods. The procedures are the same as those for getting the contents of the root MdmSchema.

Getting the Measure MdmDimension and Its Contents

The following code gets the measure MdmDimension that is in the root MdmSchema. Use this method only on the root MdmSchema. It makes no sense to use

it on subschemas, because only the root MdmSchema has a measure MdmDimension.

Example 4–5 Getting the MdmMeasureDimension and Its Contents

```
MdmMeasureDimension mdmMeasureDim = root.getMeasureDimension();
```

The following code prints the names of the MdmMeasure objects that are elements of the measure MdmDimension.

```
MdmMeasureMemberType mdMemberType =
   (MdmMeasureMemberType) mdmMeasureDim.getMemberType();
List mdList = mdMemberType.getMeasures();
Iterator mdIter = mdList.iterator();
   while (mdIter.hasNext())
      System.out.println("************Contains Measure: " +
         ((MdmMeasure) mdIter.next()).getName());
```

Getting the Characteristics of Metadata Objects

Having discovered the list of MdmMeasure and MdmDimension objects, the next step in metadata discovery involves finding out the characteristics of those objects.

Getting the MdmDimension Objects for an MdmMeasure

A primary characteristic of an MdmMeasure is that it has related MdmDimension objects. The following code gets a List of MdmDimension objects for an MdmMeasure called sales.

```
List dimsOfSales = mdmSalesAmount.getDimensions();
```

The getMeasureInfo method in the sample code provided later in this chapter shows one way to iterate through the MdmDimension objects belonging to a given MdmMeasure.

Getting the Related Objects for an MdmDimension

An MdmDimension has related MdmDimensionDefinition and MdmDimensionMemberType objects, which you can obtain by calling its getDefinition and getMemberType methods. If it is an MdmHierarchy, it also has regions, which you can obtain by calling the getRegions method on its MdmUnionDimensionDefinition.

The following is an example of how you can get the level `MdmHierarchy` objects for a union `MdmHierarchy`. The following code prints the names of the level `MdmHierarchy` objects.

```
MdmUnionDimensionDefinition unionDef =
  (MdmUnionDimensionDefinition) mdmDimObj.getDefinition();
List hierarchies = unionDef.getRegions();
for (Iterator iterator = hierarchies.iterator();
  iterator.hasNext();)
    {
    MdmHierarchy hier = (MdmHierarchy) iterator.next();
    System.out.println("Hierarchy: " + hier.getName());
   }
```

The `getDimInfo` method in the sample code provided later in this chapter shows one way to get the following metadata objects for a given `MdmDimension`:

- Its `MdmDimensionMemberType`
- Its `MdmAttribute` objects
- Its concrete class and hierarchy type
- Its parent, ancestors, and region attributes
- Its `MdmDimensionDefinition`
- Its regions. That is, if it is a union `MdmHierarchy`, the code obtains its component `MdmHierarchy` objects. If it is a level `MdmHierarchy`, the code obtains its component `MdmLevel` objects
- Its default level `MdmHierarchy`, if it is a union `MdmHierarchy`.

Methods are also available for obtaining other `MdmDimension` characteristics. See the OLAP API Javadoc for descriptions of all the methods on the MDM classes.

Getting the Source for a Metadata Object

A metadata object represents a set of data, but it does not provide the ability to create queries on that data. Its function is informational, recording the existence, structure, and characteristics of the data. It does not give access to the data values.

In order to access the data values for a given metadata object, an application gets the `Source` object that represents its data. A `Source` that represents the data for a metadata object is called a primary `Source`.

To get the primary Source for a metadata object, an application calls the getSource method on that metadata object. For example, if an application needs to display the sales figures for 1999, it must first use the getSource method on the MdmMeasure called mdmSalesAmount.

Example 4–6 Getting a Primary Source for a Metadata Object

```
Source salesAmount = mdmSalesAmount.getSource();
```

An application can call the getSource method on any object that is an instance of a concrete subclass of MdmSource. The following is a list of the concrete subclasses:

- MdmHierarchy
- MdmLevel
- MdmListDimension
- MdmAttribute
- MdmMeasure

For more information about getting and working with primary Source objects, see Chapter 5, "Introduction to Querying"

Sample Code for Discovering Metadata

The sample code that follows is a simple Java program called SampleMetadataDiscoverer. The program discovers the metadata objects that are under the root MdmSchema of any data store. The program's output lists the names and related objects for the MdmMeasure and MdmDimension objects in the root MdmSchema and its subschemas.

After presenting the program code, this topic presents the output of the program when it is run against a data store that consists of the Sales History relational schema, which is provided with the Oracle installation. In the OLAP metadata, the Sales History schema is represented as the SH_CAT measure folder. Through an OLAP API connection, the SH_CAT measure folder maps to an MdmSchema that is also called SH_CAT.

The SampleMetadataDiscoverer program includes one piece of code that is specific to the SH_CAT MdmSchema. This code gets the primary Source for an MdmDimension for which the return value of the getName method is PRODUCTS_DIM.

In most cases, an application will not search for a metadata object using its internal name (such as PRODUCTS_DIM), and it will not use the `System.out.println` method to produce output. However, this sample code uses these techniques because they offer the advantage of simplicity.

Code for the SampleMetadataDiscoverer Program

To establish a connection, this program calls a hypothetical method called `connectOnLab1` on a hypothetical class called `MyConnection`. To close the connection, the program calls a method called `MyConnection.closeConnection`. The code for these methods is not shown here, but the procedure for connecting is described in Chapter 3, "Connecting to a Data Store".

Example 4–7 Discovering the Available Metadata

```
package mytestpackage;
import com.sun.java.util.collections.ArrayList;
import com.sun.java.util.collections.List;
import com.sun.java.util.collections.Iterator;

import oracle.express.mdm.*;
import oracle.olapi.metadata.MetadataObject;

import oracle.olapi.data.source.Source;
import oracle.express.olapi.data.full.ExpressDataProvider;

public class SampleMetadataDiscoverer {

    static final int TERSE = 0;
    static final int VERBOSE = 1;

  public SampleMetadataDiscoverer(){
    }

  public static void main(String[] args) {

    // Connect through JDBC to a database on Lab1
    //     and get a DataProvider (see Chapter 3)
    ExpressDataProvider dp = MyConnection.connectOnLab1();
```

```
  // Create an MdmMetadataProvider
  MdmMetadataProvider mp = null;
  mp = (MdmMetadataProvider) dp.getDefaultMetadataProvider();

  // Get metadata info about the root MdmSchema and its subschemas
  MdmSchema root = null;
  try {
      root = mp.getRootSchema();
      System.out.println("***Root MdmSchema: " + root.getName());
      MdmDimension measureDim = root.getMeasureDimension();
      System.out.println("******Measure MdmDimension: " +
         measureDim.getName());
      getSchemaInfo(root, TERSE);
  } catch (Exception e) {
      System.out.println("***Exception encountered : " + e.toString());
  }

  // Make a Source object out of the PRODUCTS_DIM MdmDimension
  System.out.println("***Making a Source object for PRODUCTS_DIM");

  MdmDimension mdmProductDim = null;
  try {
      List rootDims = root.getDimensions();
      Iterator rootDimIter = rootDims.iterator();
      while (mdmProductDim == null && rootDimIter.hasNext()) {
         MdmDimension aDim = (MdmDimension) rootDimIter.next();
         if (aDim.getName().equals("PRODUCTS_DIM"))
           mdmProductDim = aDim;
      }
      Source product = mdmProductDim.getSource();
      System.out.println("******Made the Source");
  } catch (Exception e) {
      System.out.println("******Exception encountered : " + e.toString());
  }

  // Close the connection
  MyConnection.closeConnection(conn);
}
```

```
// ********************************************************

  // Method for getting info about an MdmSchema
  public static void getSchemaInfo(MdmSchema schema, int outputStyle) {

    System.out.println("***Schema: " + schema.getName());
    // Get the MdmSchema's dimension info
    MdmDimension oneDim = null;
    try {
        List dims = schema.getDimensions();
        Iterator dimIter = dims.iterator();
        System.out.println("   ");
        System.out.println("*********************************************");
        System.out.println("   ");
       while (dimIter.hasNext()) {
           oneDim = (MdmDimension) dimIter.next();
           getDimInfo(oneDim, outputStyle);
           System.out.println("   ");
           System.out.println("*********************************************");
           System.out.println("   ");
           }
     } catch (Exception e) {
        System.out.println("******Exception encountered : " + e.toString());
    }

    // Get the MdmSchema's measure info
    MdmMeasure oneMeasure = null;
    try {
        List measures = schema.getMeasures();
        Iterator measIter = measures.iterator();
        while (measIter.hasNext()) {
            oneMeasure = (MdmMeasure) measIter.next();
            getMeasureInfo(oneMeasure, outputStyle);
            System.out.println("   ");
            System.out.println("   ");
        }
     } catch (Exception e) {
        System.out.println("******Exception encountered : " + e.toString());
    }

    // Get the MdmSchema's subschema info
    MdmSchema oneSchema = null;
    try {
        List subSchemas = schema.getSubSchemas();
        Iterator subSchemaIter = subSchemas.iterator();
```

```
        while (subSchemaIter.hasNext()) {
           oneSchema = (MdmSchema) subSchemaIter.next();
              getSchemaInfo(oneSchema, VERBOSE);
        }
    } catch (Exception e) {
        System.out.println("***Exception encountered : " + e.toString());
    }
  }

// *********************************************************

  // Method for getting info about an MdmDimension
  public static void getDimInfo(MdmDimension dim, int outputStyle) {

    System.out.println("******MdmDimension Name: " + dim.getName());
    System.out.println("*********Description: " + dim.getDescription());

    if (outputStyle == VERBOSE) {

    // Get MdmDimensionMemberType for the MdmDimension
    try {
    MdmDimensionMemberType dimMemberType = dim.getMemberType();
    if (dimMemberType instanceof MdmStandardMemberType)
       System.out.println("*********Member Type: MdmStandardMemberType");
    if (dimMemberType instanceof MdmTimeMemberType)
       System.out.println("*********Member Type: MdmTimeMemberType");
    if (dimMemberType instanceof MdmMeasureMemberType)
       System.out.println("*********Member Type: MdmMeasureMemberType");
    } catch (Exception e) {
        System.out.println("***Exception encountered : " + e.toString());
    }

    // Get attributes of the MdmDimension
    try {
        List attributes = dim.getAttributes();
        Iterator attrIter = attributes.iterator();
        while (attrIter.hasNext())
           System.out.println("*********Attribute: " +
               ((MdmAttribute) attrIter.next()).getName());
    } catch (Exception e) {
        System.out.println("***Exception encountered : " + e.toString());
    }
```

```
// Get concrete class and hierarchy type of the MdmDimension
String kindOfDim = null;
try {
   if (dim instanceof MdmListDimension) {
      kindOfDim = "ListDim";
      System.out.println("*********" + dim.getName() +
               " is an MdmListDimension");
      }
   else if (dim instanceof MdmHierarchy)
      switch(((MdmHierarchy) dim).getHierarchyType()) {
         case (MdmHierarchy.UNION_HIERARCHY):
            kindOfDim = "UnionHier";
            System.out.println("*********" + dim.getName() +
               " is a union MdmHierarchy");
            break;
         case (MdmHierarchy.LEVEL_HIERARCHY):
            kindOfDim = "LevelHier";
            System.out.println("*********" + dim.getName() +
               " is a level MdmHierarchy");
            break;
         case (MdmHierarchy.VALUE_HIERARCHY):
            kindOfDim = "ValueHier";
            System.out.println("*********" + dim.getName() +
               " is a value MdmHierarchy");
            break;
     }
   else {
      kindOfDim = "Level";
      System.out.println("*********" + dim.getName() + " is an MdmLevel");
      }
} catch (Exception e) {
    System.out.println("***Exception encountered : " + e.toString());
}

// For level MdmHierarchy, get parent, ancestors, and region attributes
  if (kindOfDim.equals("LevelHier"))
   {
   System.out.println("*********Parent attribute: " +
      ((MdmHierarchicalDimension) dim).getParentRelation().getName());
   System.out.println("*********Ancestors attribute: " +
      ((MdmHierarchicalDimension) dim).getAncestorsRelation().getName());
   System.out.println("*********Region attribute: " +
      ((MdmUnionDimensionDefinition) dim.getDefinition())
      .getRegionAttribute().getName());
   }
```

```
   // Get the MdmDimensionDefinition for the MdmDimension
   MdmDimensionDefinition dimDef = dim.getDefinition();
   // For union or level MdmHierarchy, list the regions and default hierarchy
   if ((kindOfDim.equals("UnionHier")) || (kindOfDim.equals("LevelHier")))
      {
      try {
         System.out.println("   ");
         System.out.println("*********The following are the regions of " +
            dim.getName());
         List regions = ((MdmUnionDimensionDefinition)dimDef).getRegions();
         Iterator regIter = regions.iterator();
         while (regIter.hasNext()) {
            MdmDimension oneRegion = (MdmDimension) regIter.next();
            System.out.println("************" + oneRegion.getName());
          if (oneRegion.hasMdmTag(MdmMetadataProvider.DEFAULT_HIERARCHY_TAG))
            System.out.println("***************(The " + oneRegion.getName() +
                 " region is the default MdmHierarchy)");
         }
      } catch (Exception e) {
          System.out.println("***Exception encountered : " + e.toString());
      }
   }

   // For union or level MdmHierarchy, get region info
   if ((kindOfDim.equals("UnionHier")) || (kindOfDim.equals("LevelHier")))
      {
      try {
         System.out.println("   ");
         System.out.println("*********Information about the regions of " +
            dim.getName() + ":");
         List regions = ((MdmUnionDimensionDefinition)dimDef).getRegions();
         Iterator regIter = regions.iterator();
         while (regIter.hasNext()) {
            MdmDimension oneRegion = (MdmDimension) regIter.next();
            getDimInfo(oneRegion, VERBOSE);
            }
      } catch (Exception e) {
          System.out.println("***Exception encountered : " + e.toString());
      }
   }
  }
 System.out.println("   ");
 }
```

```
// *********************************************************

  // Method for getting info about an MdmMeasure
  public static void getMeasureInfo(MdmMeasure measure, int outputStyle) {
    System.out.println("******Measure: " + measure.getName());
    if (outputStyle == VERBOSE) {

    // Get the dimensions of the MdmMeasure
    try {
        List mDims = measure.getDimensions();
        Iterator mDimIter = mDims.iterator();
        while (mDimIter.hasNext())
           System.out.println("*********Dimension of the Measure: " +
              ((MdmDimension) mDimIter.next()).getName());
     } catch (Exception e) {
        System.out.println("******Exception encountered : " + e.toString());
    }
  }
  }
}
```

Output from the SampleMetadataDiscoverer Program

The output from the sample program consists of text lines produced by Java statements such as the following one.

```
System.out.println("***Root MdmSchema: " + root.getName());
```

The code uses the `getName` method because its return value is brief. An alternative would be to use the `getDescription` method, but the output would be more verbose.

When the program is run on the Sales History schema, the output includes the following items:

- The name of the root `MdmSchema`, which is ROOT.
- The name of the measure `MdmDimension` for the root `MdmSchema`. The name is MEASUREDIMENSION.
- The names and descriptions of the `MdmDimension` objects in the root `MdmSchema`.

- Names, descriptions, and additional information about the `MdmDimension` and `MdmMeasure` objects in the SH_CAT `MdmSchema`.

  Because the `SH_CAT` `MdmSchema` is the only subschema under the root `MdmSchema`, its `MdmDimension` objects are identical to those in the root.

- Two lines that indicate that the code got the primary `Source` for the `MdmDimension` that has the name PRODUCTS_DIM.

Here is the output. In order to conserve space, some blank lines have been omitted.

```
***Root MdmSchema: ROOT
******Measure MdmDimension: MEASUREDIMENSION
***Schema: ROOT

********************************************
******MdmDimension Name: CHANNELS_DIM
*********Description: Channel Values

**********************************************
******MdmDimension Name: CUSTOMERS_DIM
*********Description: Customer Dimension Values

**********************************************
******MdmDimension Name: PRODUCTS_DIM
*********Description: Product Dimension Values

**********************************************
******MdmDimension Name: PROMOTIONS_DIM
*********Description: Promotion Values

**********************************************
******MdmDimension Name: TIMES_DIM
*********Description: Time Dimension Values

**********************************************

***Subschema: SH_CAT
***Schema: SH_CAT

********************************************

******MdmDimension Name: CHANNELS_DIM
*********Description: Channel Values
*********Member Type: MdmStandardMemberType
*********Attribute: Long Description
```

```
*********Attribute: Short Description
*********CHANNELS_DIM is a union MdmHierarchy

*********The following are the regions of CHANNELS_DIM
************CHANNEL_ROLLUP
***************(The CHANNEL_ROLLUP region is the default MdmHierarchy)

*********Information about the regions of CHANNELS_DIM:
******MdmDimension Name: CHANNEL_ROLLUP
*********Description: Standard Channels
*********Member Type: MdmStandardMemberType
*********CHANNEL_ROLLUP is a level MdmHierarchy
*********Parent attribute: PARENTRELATION
*********Ancestors attribute: ANCESTORSRELATION
*********Region attribute: LEVELRELATION

*********The following are the regions of CHANNEL_ROLLUP
************CHANNEL_TOTAL
************CHANNEL_CLASS
************CHANNEL

*********Information about the regions of CHANNEL_ROLLUP:
******MdmDimension Name: CHANNEL_TOTAL
*********Description: Channel Total for the standard hierarchy
*********Member Type: MdmStandardMemberType
*********CHANNEL_TOTAL is an MdmLevel

******MdmDimension Name: CHANNEL_CLASS
*********Description: Channel Class level of the standard hierarchy
*********Member Type: MdmStandardMemberType
*********CHANNEL_CLASS is an MdmLevel

******MdmDimension Name: CHANNEL
*********Description: Channel level of the standard hierarchy
*********Member Type: MdmStandardMemberType
*********CHANNEL is an MdmLevel

**********************************************

******MdmDimension Name: CUSTOMERS_DIM
*********Description: Customer Dimension Values
*********Member Type: MdmStandardMemberType
*********Attribute: Long Description
*********Attribute: Short Description
*********Attribute: First Name
```

```
*********Attribute: Last Name
*********Attribute: Gender
*********Attribute: Marital Status
*********Attribute: Year of Birth
*********Attribute: Income Level
*********Attribute: Credit Limit
*********Attribute: Street Address
*********Attribute: Postal Code
*********Attribute: Phone Number
*********Attribute: E-mail
*********CUSTOMERS_DIM is a union MdmHierarchy

*********The following are the regions of CUSTOMERS_DIM
************GEOG_ROLLUP
***************(The GEOG_ROLLUP region is the default MdmHierarchy)
************CUST_ROLLUP

*********Information about the regions of CUSTOMERS_DIM:
******MdmDimension Name: GEOG_ROLLUP
*********Description: Standard
*********Member Type: MdmStandardMemberType
*********GEOG_ROLLUP is a level MdmHierarchy
*********Parent attribute: PARENTRELATION
*********Ancestors attribute: ANCESTORSRELATION
*********Region attribute: LEVELRELATION

*********The following are the regions of GEOG_ROLLUP
************GEOG_TOTAL
************REGION
************SUBREGION
************COUNTRY
************STATE
************CITY
************CUSTOMER

*********Information about the regions of GEOG_ROLLUP:
******MdmDimension Name: GEOG_TOTAL
*********Description: Geography Total for the standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********GEOG_TOTAL is an MdmLevel

******MdmDimension Name: REGION
*********Description: Region level of the standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********REGION is an MdmLevel
```

```
******MdmDimension Name: SUBREGION
*********Description: Subregion level of the standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********SUBREGION is an MdmLevel

******MdmDimension Name: COUNTRY
*********Description: Country level of the standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********COUNTRY is an MdmLevel

******MdmDimension Name: STATE
*********Description: State level of the standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********STATE is an MdmLevel

******MdmDimension Name: CITY
*********Description: City level of the standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********CITY is an MdmLevel

******MdmDimension Name: CUSTOMER
*********Description: Customer level of standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********CUSTOMER is an MdmLevel


******MdmDimension Name: CUST_ROLLUP
*********Description: Standard
*********Member Type: MdmStandardMemberType
*********CUST_ROLLUP is a level MdmHierarchy
*********Parent attribute: PARENTRELATION
*********Ancestors attribute: ANCESTORSRELATION
*********Region attribute: LEVELRELATION

*********The following are the regions of CUST_ROLLUP
************CUST_TOTAL
************STATE
************CITY
************CUSTOMER
```

```
*********Information about the regions of CUST_ROLLUP:
******MdmDimension Name: CUST_TOTAL
*********Description: Customer Total for the standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********CUST_TOTAL is an MdmLevel

******MdmDimension Name: STATE
*********Description: State level of the standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********STATE is an MdmLevel

******MdmDimension Name: CITY
*********Description: City level of the standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********CITY is an MdmLevel

******MdmDimension Name: CUSTOMER
*********Description: Customer level of standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********CUSTOMER is an MdmLevel


**********************************************

******MdmDimension Name: PRODUCTS_DIM
*********Description: Product Dimension Values
*********Member Type: MdmStandardMemberType
*********Attribute: Long Description
*********Attribute: Short Description
*********PRODUCTS_DIM is a union MdmHierarchy

*********The following are the regions of PRODUCTS_DIM
************PROD_ROLLUP
***************(The PROD_ROLLUP region is the default MdmHierarchy)

*********Information about the regions of PRODUCTS_DIM:
******MdmDimension Name: PROD_ROLLUP
*********Description: Standard
*********Member Type: MdmStandardMemberType
*********PROD_ROLLUP is a level MdmHierarchy
*********Parent attribute: PARENTRELATION
*********Ancestors attribute: ANCESTORSRELATION
*********Region attribute: LEVELRELATION
```

```
*********The following are the regions of PROD_ROLLUP
************PROD_TOTAL
************CATEGORY
************SUBCATEGORY
************PRODUCT

*********Information about the regions of PROD_ROLLUP:
******MdmDimension Name: PROD_TOTAL
*********Description: Product Total for the standard PRODUCT hierarchy
*********Member Type: MdmStandardMemberType
*********PROD_TOTAL is an MdmLevel

******MdmDimension Name: CATEGORY
*********Description: Category level of standard PRODUCT hierarchy
*********Member Type: MdmStandardMemberType
*********CATEGORY is an MdmLevel

******MdmDimension Name: SUBCATEGORY
*********Description: Sub-category level of standard PRODUCT hierarchy
*********Member Type: MdmStandardMemberType
*********SUBCATEGORY is an MdmLevel

******MdmDimension Name: PRODUCT
*********Description: Product level of standard PRODUCT hierarchy
*********Member Type: MdmStandardMemberType
*********PRODUCT is an MdmLevel


**********************************************

******MdmDimension Name: PROMOTIONS_DIM
*********Description: Promotion Values
*********Member Type: MdmStandardMemberType
*********Attribute: Long Description
*********Attribute: Short Description
*********PROMOTIONS_DIM is a union MdmHierarchy

*********The following are the regions of PROMOTIONS_DIM
************PROMO_ROLLUP
***************(The PROMO_ROLLUP region is the default MdmHierarchy)
```

```
*********Information about the regions of PROMOTIONS_DIM:
******MdmDimension Name: PROMO_ROLLUP
*********Description: Standard Promotions
*********Member Type: MdmStandardMemberType
*********PROMO_ROLLUP is a level MdmHierarchy
*********Parent attribute: PARENTRELATION
*********Ancestors attribute: ANCESTORSRELATION
*********Region attribute: LEVELRELATION

*********The following are the regions of PROMO_ROLLUP
************PROMO_TOTAL
************CATEGORY
************SUBCATEGORY
************PROMO

*********Information about the regions of PROMO_ROLLUP:
******MdmDimension Name: PROMO_TOTAL
*********Description: Promotions Total for the standard PROMOTION hierarchy
*********Member Type: MdmStandardMemberType
*********PROMO_TOTAL is an MdmLevel

******MdmDimension Name: CATEGORY
*********Description: Category level of the standard PROMOTION hierarchy
*********Member Type: MdmStandardMemberType
*********CATEGORY is an MdmLevel

******MdmDimension Name: SUBCATEGORY
*********Description: Sub-category level of the standard PROMOTION hierarchy
*********Member Type: MdmStandardMemberType
*********SUBCATEGORY is an MdmLevel

******MdmDimension Name: PROMO
*********Description: Promotion level of the standard PROMOTION hierarchy
*********Member Type: MdmStandardMemberType
*********PROMO is an MdmLevel
```

```
**********************************************

******MdmDimension Name: TIMES_DIM
*********Description: Time Dimension Values
*********Member Type: MdmStandardMemberType
*********Attribute: Long Description
*********Attribute: Short Description
*********Attribute: Period Number
*********Attribute: Period Number of Days
*********Attribute: Period End Date
*********TIMES_DIM is a union MdmHierarchy

*********The following are the regions of TIMES_DIM
************CAL_ROLLUP
***************(The CAL_ROLLUP region is the default MdmHierarchy)
************FIS_ROLLUP

*********Information about the regions of TIMES_DIM:
******MdmDimension Name: CAL_ROLLUP
*********Description: Calendar
*********Member Type: MdmStandardMemberType
*********CAL_ROLLUP is a level MdmHierarchy
*********Parent attribute: PARENTRELATION
*********Ancestors attribute: ANCESTORSRELATION
*********Region attribute: LEVELRELATION

*********The following are the regions of CAL_ROLLUP
************YEAR
************QUARTER
************MONTH
************DAY

*********Information about the regions of CAL_ROLLUP:
******MdmDimension Name: YEAR
*********Description: Year level of the Calendar hierarchy
*********Member Type: MdmStandardMemberType
*********YEAR is an MdmLevel

******MdmDimension Name: QUARTER
*********Description: Quarter level of the Calendar hierarchy
*********Member Type: MdmStandardMemberType
*********QUARTER is an MdmLevel
```

```
******MdmDimension Name: MONTH
*********Description: Month level of the Calendar hierarchy
*********Member Type: MdmStandardMemberType
*********MONTH is an MdmLevel

******MdmDimension Name: DAY
*********Description: Day level of the Calendar hierarchy
*********Member Type: MdmStandardMemberType
*********DAY is an MdmLevel


******MdmDimension Name: FIS_ROLLUP
*********Description: Fiscal
*********Member Type: MdmStandardMemberType
*********FIS_ROLLUP is a level MdmHierarchy
*********Parent attribute: PARENTRELATION
*********Ancestors attribute: ANCESTORSRELATION
*********Region attribute: LEVELRELATION

*********The following are the regions of FIS_ROLLUP
************FIS_YEAR
************FIS_QUARTER
************FIS_MONTH
************FIS_WEEK
************DAY

*********Information about the regions of FIS_ROLLUP:
******MdmDimension Name: FIS_YEAR
*********Description: Year level of the Fiscal hierarchy
*********Member Type: MdmStandardMemberType
*********FIS_YEAR is an MdmLevel

******MdmDimension Name: FIS_QUARTER
*********Description: Quarter level of the Fiscal hierarchy
*********Member Type: MdmStandardMemberType
*********FIS_QUARTER is an MdmLevel

******MdmDimension Name: FIS_MONTH
*********Description: Month level of the Fiscal hierarchy
*********Member Type: MdmStandardMemberType
*********FIS_MONTH is an MdmLevel
```

```
******MdmDimension Name: FIS_WEEK
*********Description: Week level of the Fiscal hierarchy
*********Member Type: MdmStandardMemberType
*********FIS_WEEK is an MdmLevel

******MdmDimension Name: DAY
*********Description: Day level of the Calendar hierarchy
*********Member Type: MdmStandardMemberType
*********DAY is an MdmLevel

**********************************************

******Measure: SALES_QUANTITY
*********Dimension of the Measure: CHANNELS_DIM
*********Dimension of the Measure: CUSTOMERS_DIM
*********Dimension of the Measure: PRODUCTS_DIM
*********Dimension of the Measure: PROMOTIONS_DIM
*********Dimension of the Measure: TIMES_DIM

******Measure: SALES_AMOUNT
*********Dimension of the Measure: CHANNELS_DIM
*********Dimension of the Measure: CUSTOMERS_DIM
*********Dimension of the Measure: PRODUCTS_DIM
*********Dimension of the Measure: PROMOTIONS_DIM
*********Dimension of the Measure: TIMES_DIM

******Measure: UNIT_PRICE
*********Dimension of the Measure: PRODUCTS_DIM
*********Dimension of the Measure: TIMES_DIM

******Measure: UNIT_COST
*********Dimension of the Measure: PRODUCTS_DIM
*********Dimension of the Measure: TIMES_DIM

***Making a Source object for PRODUCTS_DIM
******Made the Source
```

5

Introduction to Querying

This chapter introduces `Source` objects which are specifications for sets of data that represent the result of queries. Chapter 6, "Making Queries Using Source Methods" provides task-oriented discussions of using `Source` methods to make queries. Using `Template` objects to make queries is discussed in Chapter 10, "Creating Dynamic Queries".

This chapter includes the following topics:

- Characteristics of Source Objects
- Creating Source Objects

Characteristics of Source Objects

The OLAP API data model is unique. It is not exactly like the relational model or multidimensional model. In the OLAP API, specifications for sets of data that represent the result of queries are represented by instances of the Source class or its subclasses outlined in Table 5–1. Each Source has a paired SourceDefinition that defines the operations that created the query.

Table 5–1 Subclasses of the Source Class

Subclass	Java Type of Values	OLAP API Data Type
BooleanSource	boolean values	Boolean
DateSource	Java Date objects	Date
NumberSource	double, float, int, or short values, or some combination of these numerical values	Double, Float, Integer, Short, or Number
StringSource	Java String objects	String

Source objects are immutable. You cannot change a Source object once it has been created. When you want to present a Source object as changeable to your users, use a Source object defined by a Template object. Template objects have state and can be modified at any time. Using Template objects to make queries is discussed in Chapter 10, "Creating Dynamic Queries".

Source objects are only specifications for a data set; they do not themselves actually have data. Even so, it helps to think of them as the result set they define. From this perspective, a Source knows its type and structure (inputs and outputs).

Source Type

All Source objects have type. In the OLAP API, the type of a Source is another Source from which the Source obtains its values. You can retrieve the type of a Source using the getType method.

The OLAP API provides a FundamentalMetadataObject to represent each of the fundamental Java data type, the Java String object, and the Java Date object. These objects are known as the OLAP API data types. The OLAP API data types of Source objects and their relationship to each other are shown in Table 5–2. You can create a Source object that represents an OLAP API data type by following the process outlined in "Creating Source Objects that Represent OLAP API Data Types" on page 5-9. You can retrieve the OLAP API type of a Source using the getDataType method.

The operation that creates a new Source often determines the type of that Source. For example, assume that you have a Source object named customer whose values are the unique numerical identifier for each customer. The OLAP API type of customer is Integer. Assume, additionally, that you use the select method on customer to create another Source object named customerSelection. The OLAP API type of customerSelection is customer.

Table 5–2 OLAP API Data Types of Source Objects

OLAP API Data Type	Descriptions
Value	A Source object with any OLAP API data type.
Boolean	A Source object whose values have the Java boolean data type.
Date	A Source object whose values are Java Date objects.
Number	A Source object with any of OLAP API numerical data type.
Double	A Source object whose values have the Java double data type.
Float	A Source object whose values have the Java float data type.
Integer	A Source object whose values have the Java int data type.
Short	A Source object whose values have the Java short data type.
String	A Source object whose values are Java String objects.
Empty	A Source object that does not have any values defined for it.
Null	A Source object that has a single null value.

Source Structure: Inputs and Outputs

All Source objects (except for an empty Source) have values. In some cases, the values of a Source are unique data items that are meaningful unto themselves. If you are familiar with relational concepts, you can conceptualize this type of Source as a table with a single column -- the column that contains the values of the Source. If you are more familiar with multidimensional concepts, you can conceptualize this type of Source as a dimension.

In other cases, the values of a Source are not unique data items and, thus, are not meaningful unto themselves. Instead the values of the Source are meaningful only in relationship to the values of another Source. In this case, the structure of the Source is determined by other Source objects called inputs and outputs. Whether

one of these other Source objects is an input or an output is determined by whether or not values have been specified for it:

- **Outputs.** When values have been specified, the other Source object is called an output. The values of a Source are identified by the set of its output values. You can retrieve the outputs of a Source using the getOutputs method.
- **Inputs.** When values have *not* been specified, the other Source object is called an input. A Source that has inputs is an indeterminable result set. If you are familiar with relational concepts, you think of an input as a column that acts as a key to the values of a Source, but that is in the GROUP BY list of a SQL statement. If you are more familiar with multidimensional concepts, you can conceptualize an input that is a dimension of a Source, but that is not in its dimension list. You can retrieve the inputs of a Source using the getInputs method.

The inputs and outputs of a Source determine how the Source is processed by Oracle OLAP. For example, when a Cursor is opened on a Source, Oracle OLAP loops over its outputs in order to produce the data, but it (arbitrarily) qualifies away any of its inputs. In order to retrieve one or more values of a Source with inputs, you must specify the values for its inputs that will uniquely identify the desired values of the Source. The order in which you specify values for the inputs determines the structure and processing of a Source. The input that you specify values for first becomes the slowest-varying output. For more information on specifying values for inputs, see "Selecting Based on Output Values" on page 6-3 and "Effect of Input-Output Order on Source Structure" on page 6-6.

Additionally, when a Source has both inputs and outputs, the values of the Source are identified by the set of its output value and each set of possible output values typically identifies a number of values (that is, a subset of data). Some Source methods work on these subsets of data. For example, Oracle OLAP loops over the outputs of a Source when it processes any methods that select values based on their positions (the first value of each subset has a position of 1) or any aggregation methods like average and total. For an in-depth discussions of the positional and aggregation methods, see "Finding the Position of Values" on page 6-6 and "Working with Aggregation Methods" on page 6-24,

Creating Source Objects

Making queries using the OLAP API is a process that involves creating a number of different Source objects. This process is outlined below:

1. Create Source objects that correspond to metadata objects as described in "Getting Source Objects From Metadata Objects" on page 5-2. These Source objects, sometimes called **primary** Source objects, have a structure that is similar to the metadata objects from which they are created.
2. Begin your querying by calling methods on the primary Source objects. The methods of the Source class and its subclasses return new Source objects sometimes called **derived** Source objects. Continue your analysis by deriving additional Source objects until you have the results you want to retrieve the data into your program. Derived Source object and the methods that create them are introduced in "Creating New Source Objects Using Source Methods" on page 5-7 and documented in detail in the OLAP API Javadoc. Task-oriented discussions of how to use Source methods to make selections and perform typical analytic operations are provided in Chapter 6, "Making Queries Using Source Methods".

   As part of the query process, you might also create simple nondimensional Source object to use as operands when making selections and calculations and Source objects that represent OLAP API data types. How to create these types of Source objects is discussed in more detail in"Creating Simple Nondimensional Source Objects" and "Creating Source Objects that Represent OLAP API Data Types" on page 5-9.
3. When you want to retrieve the data set represented by a Source object, create a Cursor for it as described in Chapter 9, "Retrieving Query Results".

Getting Source Objects From Metadata Objects

To get a Source object from a metadata object, take the following steps:

1. Create the metadata data object for which you want to create a corresponding Source object as described in Chapter 2.
2. Use the getSource method to create a Source object from the metadata object.

Creating a Source from MdmDimension, MdmHierarchy, or MdmLevel Objects

A `Source` that you create by using the calling the `getSource` method on a `MdmDimension`, an `MdmHierarchy`, or an `MdmLevel` does not have any inputs or outputs. It is a specification for a simple list of values.

In "Level MdmHierarchy for Calendar Year" on page 2-13, we created an `MdmHierarchy` named `mdmTimesDimCalHier`. To create a `Source` named `timesDimCalHier` from `mdmTimesDimCalHier`, you use code shown in Example 5–1.

Example 5–1 Getting a Source for an MdmHierarchy

```
Source timesDimCalHier = mdmTimesDimCalHier.getSource;
```

The `Source` named `timesDimCalHier` consists of a simple non-indexed list of 1529 values: 4 values for year, 16 values for quarters, 48 values for months, and 1461 values for days.

timesDimCalHier values
1998
1998-Q1
1998-01
01-JAN-98
02-JAN-98
03-JAN-98
...
1998-Q2
1998-04
01-APR-98
. . .
1999
...

Creating a Source from MdmMeasure or MdmAttribute Objects

A Source that you create by calling the getSource method on a MdmMeasure or an MdmAttribute is a specification for a data set that has one or more inputs. Each of these inputs is a primary Source that was created from a MdmDimension. Thus, the specification for a set of data represented by a Source that you create from an MdmMeasure or an MdmAttribute is incomplete. Consequently, you cannot create a Cursor on these Source objects to retrieve their values into the application. To retrieve the values of a Source created from a MdmMeasure or an MdmAttribute, you must derive a new Source from it by specifying values for the values of the Source objects that act as its dimensions as described in "Selecting Based on Output Values" on page 6-3.

In "MdmMeasure with two MdmDimension objects" on page 2-21, we created an MdmMeasure named mdmUnitCost. To create a Source named unitCost from mdmUnitCost, you use code shown in Example 5–2.

Example 5–2 Getting a Source for an MdmMeasure

```
Source unitCost = mdmUnitCost.getSource;
```

Since mdmUnitCost has mdmProductsDim and mdmTimesDim as its MdmDimension objects, unitCost has two inputs (productsDim and timesDim). In order to retrieve one or more values of unitCost, you must specify the values for its inputs (productsDim and for timesDim) that will uniquely identify the values of unitCost. For information on specifying values for inputs, see "Selecting Based on Output Values" on page 6-3.

Creating New Source Objects Using Source Methods

Most OLAP queries derive new Source objects from existing Source objects using the methods in the Source class.

Table 5–1 outlines the most important Source methods in the OLAP API.

Table 5–3 The Major Source Methods

Method	Description
join	The single most important Source creation method in the OLAP API. The join method creates a new Source by combining the values of the base Source and another Source (called the joined Source) and filtering this set of data using a third Source (called the comparison Source) in the specified manner. Using an optional parameter, you can also use the join method to add the joined Source as n output to the new Source.
alias	Creates a new Source object that is the same as the base Source object, but that has the base Source as its type.
distinct	Creates a new Source object that is the same as the base Source object, but that has all duplicate rows (tuples) removed.
position	Creates a new Source object with the same structure as the base Source and whose values are the position of the values of the base Source.
value	Creates a new Source object that has the values of the base Source and that has the base Source as an input.

The OLAP API provides various other methods that you can use instead of the methods listed in Table 5–3, " The Major Source Methods". These methods include variations on the join method, as well as methods such as appendValue, at, cumulativeInterval, first, ge, interval, selectValues, and sortAscending. All of these methods are documented in the OLAP API Javadoc. For task-oriented discussions on using these methods to analyze your data, see Chapter 6, "Making Queries Using Source Methods".

Creating Simple Nondimensional Source Objects

You create simple nondimensional Source objects which are not based on metadata objects or other Source objects that you can use as operands by using the createConstantSource, createListSource, and createRangeSource methods on the DataProvider class. These Source objects are sometimes referred to as **constant**, **list**, and **range** Source objects.

Assume that you have an object named myDataProvider that represents the DataProvider used by your application and that, for computational purposes,

you want a Source with a single value of 4. To create this Source you issue the code shown in Example 5–3

Example 5–3 Creating a Constant Source

```
NumberSource myConstantFour = myDataProvider.createConstantSource(4);
```

Creating Source Objects that Represent OLAP API Data Types

You can retrieve the objects that represent the OLAP API data types using methods on the FundamentalMetadataProvider. Each of these methods returns a FundamentalMetadataObject. The OLAP API data types and the methods you use to retrieve them are shown in Table 5–4.

Table 5–4 Methods that Retrieve Objects that Represent OLAP API Data Types

OLAP API Data Type	Method that Retrieves This Data Type
Value	getValueDataType
Boolean	getBooleanDataType
Date	getDateDataType
Number	getNumberDataType
Double	getDoubleDataType
Float	getFloatDataType
Int	getIntegerDataType
Short	getShortDataType
String	getStringDataType
Empty	getEmptyDataType To retrieve an empty Source, use the DataProvider.getEmptySource method.
Null	getVoidDataType To retrieve a null Source, use the DataProvider.getVoidSource method.

To create a Source object that represents an OLAP API data type, take the following steps:

1. Get the FundamentalMetadataProvider by using the getFundamentalMetadataProvider method on the DataProvider class.
2. Create the FundamentalMetadataObject object that represents the OLAP API data type by using the appropriate method on the FundamentalMetadataProvider class.
3. Create a Source from the objects returned in Step 1 by using the FundamentalMetadataObject.getSource method.

Example 5–4 creates a Source object called olapBooleanDataType that represents the OLAP API Boolean data type. You can use olapBooleanDataType to check to see if the OLAP API data type of any other Source is Boolean.

Example 5–4 Creating a Source for the OLAP API Boolean Data Type

```
FundamentalMetadataObject myFundamentalMetadataProvider =
    myDataProvider.getFundamentalMetadataProvider();
FundamentalMetadataObject olapBooleanFundObj =
    myFundamentalMetadataProvider.getBooleanType();
Source olapBooleanDataType = olapBooleanFundObj.getSource();
```

6 Making Queries Using Source Methods

Many queries are made by calling a `Source` method on a `Source` object to create new `Source` objects. `Source` methods are introduced in "Creating New Source Objects Using Source Methods" on page 5-7 and documented in detail in the OLAP API Javadoc. This chapter discusses how to make queries using these methods.

This chapter includes the following topics:

- Selecting Based on Source Value
- Selecting Based on Output Values
- Selecting Values Based on Rank
- Selecting Values Based on Hierarchical Position
- Creating a Source that is a Self-Relation
- Performing Numerical Analysis
- Manipulating String Values

Selecting Based on Source Value

You can create a new `Source` from an existing `Source` by selecting only certain values of the base `Source`. Typically, you use one of the following methods to select based on the values of the base values:

```
Source.select(BooleanSource), Source.selectValue(Source)
Source.selectValues(Source)
Source.selectValues(Source[])
BooleanSource.selectValue(boolean)
BooleanSource.selectValues(boolean[])
NumberSource.selectValue(double)
NumberSource.selectValue(int)
NumberSource.selectValue(float)
NumberSource.selectValue(short)
NumberSource.selectValues(double[])
NumberSource.selectValues(float[])
NumberSource.selectValues(int[])
NumberSource.selectValues(short[])
StringSource.selectValue(String)
StringSource.selectValues(String[])
```

You can also select values using the `join` method using the syntax shown below.

```
Source Source::join (Source joined,
     Source comparison,
     Source.COMPARISON_RULE_SELECT,
     boolean visible);
```

Assume that you have a primary `Source` objects named `timesDim` that you created from an `MdmDimension` object named `mdmTimesDim` and whose values are the calendar values. To select only the those values for 1996, you can issue the code shown in Example 6–1

Example 6–1 Selecting Based on Source Values

```
Source timesSel = timesDim.selectValue("1996");
```

Selecting Based on Output Values

If you want to create a Cursor on a Source object, it cannot have any inputs. Since any Source created from an MdmMeasure or an MdmAttribute has inputs, the need to specify values for inputs is so universal that the OLAP API has a special join method to support it

Specifying values for the inputs of a Source is called *changing inputs to outputs*. In this sense, moving a Source from the list of inputs returned by the getInputs method to the list of outputs returned by getOutputs is similar to moving a column out of the GROUP BY list in SQL.

Using the join Method to Change Inputs to Outputs

To specify values for the input of a Source, thereby changing an input to an output, use the following join method where the original Source is the Source object that has the input that you want to become an output and the joined Source is the input you want to change.

```
Source newSource = base.join (Source joined);
```

This is a shortcut for the following join method.

```
Source newSource = base.join (joined, emptySource, Source.COMPARISON_RULE_REMOVE, true);
```

Note that the comparison Source is the empty Source that has no values. Consequently, even though the COMPARISON_RULE_REMOVE constant is specified, no values are removed as a result of the comparison. Also, because the visible flag is set to true, the joined Source becomes an output of the new Source.

Additionally, since many of the methods of Source class and its subclasses are methods that implicitly call the join method, some of these methods also change inputs to outputs.

Effect of Input-Output Order on Source Structure

The structure of a Source is determined by the order in which you turn the inputs of the Source into outputs. For a Source that has outputs, the first output that was created is the fastest-varying output; the last output that was created is the slowest-varying output.

When you string two join methods together in a single statement, the first join (reading left to right) is processed first. Consequently, when creating a single statement containing several join methods, make sure that the input that you want

to be the fastest-varying of the new Source is the joined Source in the first join in the statement.

Assume that you have a primary Source named unitCost that you created from a MdmMeasure object named mdmUnitCost. The Source named unitCost has inputs of timesDim and productsDim, and no outputs.The timesDim and productsDim Source objects do not have any inputs or outputs. The order in which you turn the inputs of unitCost into outputs determines the structure of a Source on which you can create a Cursor. Example 6–2 shows the results when you join first to timesDim. Example 6–3 shows the results when you join first to productsDim.

Changing Inputs to Outputs with timesDim as the First Output Created

Assume also that you issue the code shown in Example 6–2 to turn the inputs of the primary Source named unitCost into outputs.

Example 6–2 Changing Inputs to Outputs with timesDim as the First Output Created

```
Source newSource = unitCost.join(timesDim).join(productsDim);
```

This code strings two join methods together. Because unitCost.join(timesDim) is processed first, the output values for timesDim are the first output values specified. You can also say that timesDim is the first output defined for the new Source. After the first join is processed, the set of data represented by the resulting unnamed Source has the structure depicted below.

timesDim (output1)	values of unitCost
1998	4,000 500
31-DEC-01	9 500

After the second join is processed, the set of data represented by newSource consists of the names and the values of both of its outputs (that is, timesDim and productsDim). Since timesDim was the first output for which values were

specified, it is the fastest-varying output and the new `Source` has the structure depicted below.

productsDim (output2)	timesDim (output1)	values of unitCost
Boys	1998	4,000
Boys	31-DEC-01	10
49780	1998	500
49780	31-DEC-01	9

Changing Inputs to Outputs with productsDim as the First Output Created

Assume that you issue the code in Example 6–3 to turn the inputs of unitCost into outputs.

Example 6–3 Changing Inputs to Outputs with productsDim as the First Output Created

```
Source newSource = unitCost.join(productsDim).join(timesDim);
```

This code shown in Example 6–3 strings two `join` methods together. Because `unitCost.join(productsDim)` is processed first, `productsDim` is the first output defined for the new `Source`. Consequently, `productsDim` is the fastest-varying output and the new `Source` has the structure depicted below.

timesDim (output2)	productsDim (output1)	values of unitCost
1998	Boys	4,000
1998	49780	500
31-DEC-01	Boys	10
31-DEC-01	48780	9

Selecting Based on Output Values and Source Values: Example

Assume that you have three primary `Source` objects named `productsDim`, `promotionsDim`, `channelsDim`, and `timesDim`, that you got from `MdmDimension` objects and that you have a primary `Source` object named `sales` that you got from an `MdmMeasure` object. The `productsDim`, `promotionsDim`, `channelsDim`, and `timesDim` objects do not have any outputs. The `sales` object has `productsDim`, `promotionsDim`, `channelsDim`, and `timesDim` as inputs.

To create a new `Source` named `bigSeller` whose values are all of the products that sold more than $10,000,000 in 1996, you can issue the code shown in Example 6–4.

Example 6–4   Selecting Based on Output Values and Source Values

```
Source promotionSel = promotionsDim.selectValue("Promo total");
Source channelSel = channelsDim.selectValue("Channel total");
Source timeSel = timesDim.selectValue("1996");
Source bigSellers = productsDim.select(sales.gt(10000000)).
    join(promotionSel).join(timeSel).join(channelSel);
```

Selecting Values Based on Rank

When a `Source` is sorted according to some attribute (or attributes), then the position of the values of the `Source` represents a kind of ranking — the so-called unique ranking. There are many other types of rankings that are not unique and that are called variant rankings.

Finding the Position of Values

You can also use the methods described in Table 6–1 to find values based on their position in a `Source` or to find the position of values with the specified value or values. In the OLAP API, position is a one-based value. As described in "Finding the Positions of Values When There are no Inputs or Outputs" on page 6-8, when a `Source` has no inputs, position works against the entire set of `Source` values and only one value has a position of one. As described in "Finding the Positions of Values When There Are Outputs and Inputs" on page 6-8, when a `Source` has

inputs, position works against the subsets of Source values identified by each unique set of output values and the first value in each subset has a position of one.

Table 6–1 Methods for Finding Values Based on Position

Method	Description
position()	Creates a new Source with the type of Integer, the base Source as an input, and with values that are the one-based position of the values of the base Source. If the base Source is sorted according to some attribute (or attributes), then the position represents a kind of ranking - the so called unique ranking
at(pos)	Creates a new Source that has the same structure as the base Source but that only has the value that is at the specified position of the base Source. There are two versions of this method. One version allows you specify the position using a Source object; in the other, you specify position using an int value.
first()	Creates a new Source that has the same structure as the base Source but that only has the value that is at position 1 of the base Source.
last()	Creates a new Source that has the same structure as the base Source but that only has the value that is at the last position of the base Source.
positionOfValue(value)	Creates a new Source that has the same structure as the base Source but that whose values are the positions of the specified value of the base Source. There are two versions of this method. One version allows you specify the value using a Source; in the other, you specify value using a String.
positionOfValues(values)	Creates a new Source that has the same structure as the base Source but whose values are the positions of the specified values of the base Source. There are two versions of this method. One version allows you specify the value using a Source; in the other, you specify value using an array of String objects.

Finding the Positions of Values When There are no Inputs or Outputs

Assume that there is a `Source` named `products` (shown below) that has no inputs or outputs and whose values are the unique identifiers of products.

values of products
395
49780

To create a new `Source` named `productsPosition` hose values are the positions of the values of `products`, issue the code shown in Example 6–5.

Example 6–5 Finding the Position of Values When There are no Inputs or Outputs

```
Source productsPosition = products.position();
```

A tabular representation of `productsPosition` showing the position of the values in `products` is shown below. Note that the `position()` method is one based.

values of products	position of values
395	1
49780	2

Finding the Positions of Values When There Are Outputs and Inputs

Assume that there is a `Source` named `unitsSoldByCountry` (shown below) that has an output of `products`, an input of `countries`, and whose values are the total number of units for each product sold for each country.

products (output)	values of unitsSoldByCountry
395	500 800
49780	10000 50

To create a new `Source` named `positionUnitsSoldByCountry` whose values are the positions of the values of `unitsSoldByCountry`, issue the code in Example 6–6.

Example 6–6 Finding the Position of Values When there are Outputs and Inputs

```
Source positionUnitsSoldByCountry = unitsSoldByCountry.position();
```

A tabular representation of `positionUnitsSoldbyCountry` showing the position of values on `unitsSoldByCountry` is shown below.

products (output)	values of positionUnitsSoldbyCountry
395	1 2
49780	1 2

Values Ranked in Ascending or Descending Order

One of the simplest kinds of ranking is to sort the values of a `Source` in ascending or descending order.

Example 6–7 creates a new `Source` named `sortedTuples` whose values are the same as the `Source` named `base` in sorted ascending order. Example 6–8 ranks the values of the `Source` named `base` in descending order.

Example 6–7 Ranking Values in Ascending Order

```
Source sortedTuples = base.sortAscending();
```

Example 6–8 Ranking Values in Descending Order

```
Source sortedTuples = base.sortDescending();
```

Values Ranked in the Same or the Opposite Order as the Values of Another Source

You can rank the values of a `Source` by sorting them in the same or the opposite order of the values of another `Source`.

Example 6–9 creates a new ranks the values of a `Source` named `base` in the same order as the `Source` named `sortValue`. Example 6–10 the values of a `Source` named `base` in the opposite order as the `Source` named `sortValue`.

Example 6–9 Ranking Values in the Same Order as Another Source

```
Source sortedTuples = base.sortAscending(Source sortValue);
```

Example 6–10 Ranking Values in the Opposite Order as the Values of Another Source

```
Source sortedTuples = base.sortDescending(Source sortValue);
```

Minimum Ranking

Minimum ranking differs from unique ranking (position) in the way it deals with ties (values in the `Source` that share the same value for the attribute). All ties are given the same rank, which is the minimum possible.

Example 6–11 ranks values in different ways where the `Source` (named `base`) whose values you want to rank has two inputs named `input1` and `input2`.

Example 6–11 Minimum Ranking

```
Source sortedTuples = base.join(input1).sortDescending(input2);
Source equivalentRankedTuples =
  sortedTuples.join(input2, input2);
NumberSource minRank = sortedTuples.
  positionOfValues(equivalentRankedTuples).minimum();
```

Maximum Ranking

Maximum ranking differs from unique ranking (position) in the way it deals with ties (values in the `Source` that share the same value for the attribute). All ties are given the same rank, which is the maximum possible rank.

Example 6–12 ranks values in different ways where the `Source` (named `base`) whose values you want to rank has two inputs named `input1` and `input2`.

Example 6–12 Maximum Ranking

```
Source sortedTuples = base.join(input1).sortDescending(input2);
Source equivalentRankedTuples =
   sortedTuples.join(input2, input2);
NumberSource maxRank = sortedTuples.positionOfValues
     (equivalentRankedTuples).maximum();
```

Average Ranking

Average ranking differs from unique ranking in the way it deals with ties (values in the `Source` that share the same value for the attribute). All ties are given the same rank, which is equal to the average unique rank for the tied values.

Example 6–13 code ranks values in different ways where the `Source` (named `base`) whose values you want to rank has two inputs named `input1` and `input2`.

Example 6–13 Average Ranking

```
Source sortedTuples = base.join(input1).sortDescending(input2;
Source equivalentRankedTuples =
  sortedTuples.join(input2, input2);
NumberSource averageRank = sortedTuples.positionOfValues
   (equivalentRankedTuples).average();
```

Packed Ranking

Packed ranking, also called dense ranking, is distinguished from minimum ranking by the fact that the ranks are packed into consecutive integers.

Example 6–14 ranks values in different ways where the `Source` (named `base`) whose values you want to rank has two inputs named `input1` and `input2`.

Example 6–14 Packed Ranking

```
Source tuples = base.join(output1);
Source firstEquivalentTuple = tuples.join(input2, input2.first();
Source packedRank = firstEquivalentTuple.join(tuples).
  sortDescending(input2).positionOfValues(base.value().
  join(time.value());
```

Percentile Ranking

Assume that you want to use the following formula to calculate the percentile of an attribute `A` for a `Source` S with `N` values.

```
Percentile(x) =  number of values
(for which the A differs from A(x))
that come before x in the ordering * 100 / N
```

The percentile, then, is equivalent to the `minimum rank -1 * 100 / N`.

Example 6–15 ranks values in different ways where the `Source` (named `base`) whose values you want to rank has two inputs named `input1` and `input2`.

Example 6–15 Percentile Ranking

```
Source sortedTuples = base.join(input1).sortDescending(input2);
Source equivalentRankedTuples =
  sortedTuples.join(input2, input2);
NumberSource minRank = sortedTuples.
     positionOfValues(equivalentRankedTuples).minimum();
NumberSource percentile = minRank.minus(1).times(100).
     div(sortedTuples.count());
```

nTile Ranking

nTile ranking for a given *n* is defined by dividing the ordered `Source` of size count into *n* buckets, where the bucket with rank `k` is of size. The ntile rank is equivalent to the formula `ceiling*((uniqueRank*n)/count)`.

Example 6–16 code ranks values in different ways where the `Source` (named `base`) whose values you want to rank has two inputs named `input1` and `input2`.

Example 6–16 nTile Ranking

```
NumberSource n = ...;
Source sortedTuples = base.join(input1).sortDescending(input2);
NumberSource uniqueRank = sortedTuple.
      positionOfValues(base.value().join(input1.value());
NumberSource ntile = uniqueRank.times(n).
     div(sortedTuples.count()).ceiling();
```

Selecting Values Based on Hierarchical Position

In order to select values based on their hierarchical position you need to navigate the hierarchy. To navigate within a hierarchy you need to create two primary Source objects: a primary `Source` that corresponds to the hierarchy, and a primary `Source` that represents the parent-child relationships within this hierarchy.

Creating a Primary Source that Represents a Default Hierarchy

To create a Source that represents a default hierarchy, you take the following steps:

1. Retrieve the default hierarchy of the MdmDimension by taking the following steps:
   a. Check to see if the MdmDimension is a union dimension by checking to see if it has an MdmUnionDimensionDefinition.
   b. If the MdmDimension has an MdmUnionDimensionDefinition, then check to see if it has a regions that are MdmHierarchy objects.
   c. If the MdmDimension has regions that are MdmHierarchy objects, select the MdmHierarchy that is its default hierarchy.
2. Make the default hierarchy a Source object, by calling the getSource method on it.

The getMyDefaultHierarchy retrieves the default hierarchy of an MdmDimension is shown below. This method calls the getMyRegions method that retrieves the regions of an MdmDimension which, in turn, calls the getMyMdmUnionDimensionDefinition method that checks to see if the MdmDimension is a union dimension.

Example 6–17 Retrieving a Default Hierarchy

```
// method that gets all of the Regions of an MdmDimension
private MdmHierarchy getMyDefaultHierarchy(MdmDimension mdmDim) {
      List hierarchies = getMyRegions(mdmDim);
      if ( hierarchies == null )
        return null;
      for (Iterator iterator = hierarchies.iterator(); iterator.hasNext();) {
        MdmHierarchy hier = (MdmHierarchy) iterator.next();
        if (hier.hasMdmTag(MdmMetadataProvider.DEFAULT_HIERARCHY_TAG))
          return hier;
      }
    return null;
  }
```

```
// method that gets all of the Regions of an MdmDimension
private List getMyRegions(MdmDimension mdmDimension ) {
      MdmUnionDimensionDefinition unionDimDef =
   getMyMdmUnionDimensionDefinition ( mdmDimension );
      if ( unionDimDef != null )
        return unionDimDef.getMyRegions();
    return null;
  }

// method that checks to see if MdmDimension is a UnionDimension
private MdmUnionDimensionDefinition getMyMdmUnionDimensionDefinition(
MdmDimension
      mdmDimension ) {
      MdmDimensionDefinition dimDef = mdmDimension.getDefinition();
      if((dimDef == null) || (!(dimDef instanceof MdmUnionDimensionDefinition)))
        return null;
      return (MdmUnionDimensionDefinition) dimDef;
    return null;
  }
```

Creating a Primary Source for the Parent-Child Relationship

If an `MdmHierarchy` is a level hierarchy, it's values are in parent-child relationship to each other. To create a `Source` object that represents the parent-child relationships within a hierarchy, you take the following steps:

1. Create an `MdmAttribute` that represents the parent-child relationships by using the `getParentRelation` method on the `MdmHierarchy`.
2. Create a `Source` from the MdmAttribute created in step 1 by using the `getSource` method.

Creating Source Objects for Other Relationships

A feature of the OLAP API representation of a relation, such as a parent-child relation, is that it is directional. A `Source` object that represents a parent-child relation maps the children to the parent, but not the parents to the children. By contrast, in SQL a table that represent the relationship is non-directional. The basic reason is that the OLAP API, unlike SQL, uses the structure of `Source` objects to automatically determine how they `join`. Since in the OLAP API relations are directional, if you want a relation to be in the opposite direction, you need to invert it.

Assume that there is a Source named parentChild on a hierarchy named levelHierarchy. To create Source objects that represent other relationships, you join these two Source objects in different ways. In other words, as shown in Example 6–18, Example 6–19, and Example 6–20, you can create new Source objects that represent the children, siblings, and grandparents in the hierarchy by using the join method on the Source that represents the parentChild relation. You can also drill down a hierarchy as shown in "Drilling Down a Hierarchy: Example" on page 6-15.

Example 6–18 Selecting the Children

```
Source childParent = levelHierarchy.join(parentChild,
   levelHierarchy.value());
```

Example 6–19 Selecting Siblings

```
Source siblingParent = levelHierarchy.join(parentChild, parent);
```

Example 6–20 Selecting Grandparents

```
Source grandParent = parentChild.join(levelHierarchy, parentChild);
```

Drilling Down a Hierarchy: Example

Assume that there is an MdmDimension object for which you have created a Source named productsDim. Assume also that this MdmDimension object has a default hierarchy for which you have created an MdMHierarchy called prodStdHierObj and a Source called prodHeir. Example 6–21 drills down the "Trousers - Women" division of the hierarchy.

Example 6–21 Drilling Down a Hierarchy

```
// Get the parent relation from the hierarchy
MdmAttribute prodHierParentObj = prodStdHierObj.getParentRelation();
StringSource prodHierParent = prodHierParentObj.getSource();
// Select children of Trousers - Women
// - Reverse the parent relation to get a children relation
Source prodHierChildren = prodHier.join(prodHierParent, prodHier.value());
// - Note the join is hidden because we only want the children of
// - Trousers - Women, and not Trousers - Women itself
Source trousersChildren = prodHierChildren.join(prodHier,
   context.getDataProvider().createConstantSource("Trousers - Women"), false);
```

```
// Select Shirts - Boys, Trousers - Women, and Shorts - Men
Source prodHierSel = prodHier.selectValues(new String[]
  {"Shirts - Boys","Trousers - Women","Shorts - Men"});
// Insert the children of Trousers - Women after Trousers - Women
// (which is 2nd value)
Source drilledProdHierSel = prodHierSel.appendValues(trousersChildren);
// This selection has the effect of sorting the result in hierarchical order.
Source result = prodHier.selectValues(drilledProdHierSel);
```

Creating a Source that is a Self-Relation

Suppose we want to do a region-to-region comparison in some way. Specifically, suppose we want to create a data view in which the regions appear on both the rows and the columns. In the OLAP API you use the `alias()` and the `value()` methods to do this. The `alias()` method creates a new `Source` that mirrors exactly the original `Source` in terms of its data, its inputs, and its outputs. The only difference is that the original `Source` becomes the type of the alias `Source`. The `value()` method creates a new `Source` that has the original `Source` as both its type and as an input.

Assume that there would naturally be an input-output match between input `A` of the original `Source` (called `base`) and some output `B` of the joined `Source` in the `join` shown below.

```
Source result = base.join(joined, comparison);
```

To avoid this input-output match, and hence keep `A` as an input of the result, use the code shown in Example 6–22.

Assume that we have a `Source` named `region` that does not have any inputs or outputs and whose values are the names of geographical regions. Assume also that we want to create a data view in which the regions appear on both the rows and the columns. For each cell in this table we want to show the percentage difference between the areas (in square miles) of the regions. In other words, we want to create

a `Source` named `regionComparison` that has two inputs -- both of them the `Source` named `regions`. Example 6–23 shows how you do this.

Example 6–22 Procedure for Creating a Self-Relation

```
//Create an alias Source named B2 for a Source named B;
Source B2 = B.alias();
//Create a variant of the original called base2
//We know that input A will match to B
Source base2 = base.join(B, B2.value());
//Now join base2 and joined
//We know that input B2 will not match to B in joined
Source preResult = base2.join(joined, comparison);
//Finally, join to the B2 and regain the input A
Source result = preResult.join(B2, A.value());
```

Example 6–23 Creating a Source that is a Self-Relation

```
//Create an alias for region that is for the row
Source rowRegion = region.alias();

//Create an alias for region that is for the column
Source columnRegion = region.alias();

//Create rowRegionArea which has an input of rowRegion,
//     an output of area,
//     and values whose values are the same as those of region
Source rowRegionArea = area.join(rowRegion.value());

//Create columnRegionArea which has an input of columnRegion,
//     an output of area,
//     and values whose values are the same as those of region
Source columnRegionArea = area.join(columnRegion.value());

//Compute the values of the cells
Source areaComparison = rowRegionArea.div(columnRegionArea).times(100);

//Create a new Source with outputs rather than inputs
Source regionComparison = areaComparison.join(rowRegion.join(columnRegion));
```

The first two lines of code create two new `Source` objects that are aliases for the `Source` named `region`. These `Source` objects are called `rowRegion` and `columnRegion`.

The next two lines of code create `Source` objects, named `rowRegionArea` and `columnRegionArea`, that represent the areas of `rowRegion` and `columnRegion` respectively. To create `rowRegionArea`, we `join` `area` which has the input of `region` to `rowRegion.value()` which has an input of `rowRegion` and the same values as `region`. The `rowRegionArea` `Source` has an input of `rowRegion`, an output of `area`, and values whose values are the same as those of `region`. To create `columnRegionArea`, we `join` area which has the input of `region` to `columnRegion.value()` which has an input of `columnRegion` and the same values as `region`. The `Source` named `columnRegionArea` has an input of `columnRegion`, an output of `area`, and values that are the same as those of `region`. These `join` calls have the effect of replacing the `region` input with `rowRegion` or `columnRegion`, which, since they both have the names as regions as data, makes no real difference to the value of `area`.

The next line of code performs the needed computation. Because `rowRegionArea` has `rowRegion` as an input and `columnRegionArea` has `columnRegion` as an area, the new `Source` named `areaComparison` has two inputs, `rowRegion` and `columnRegion`, both of whose values are the names of regions. What we have done is to effectively create a `Source` object that has duplicate inputs.

The final step of changing inputs to outputs is easy. We merely `join` `areaComparison` to its inputs (`rowRegion` and `columnRegion`).

Performing Numerical Analysis

The `NumberSource` class and its subclasses define methods that are numeric-specific versions of various `Source` methods that you can use to append, insert, select, and remove numeric values. The `NumberSource` class and its subclasses also have methods that you can use to perform simple numerical operations such as subtraction and division, make numerical comparisons, perform standard numerical functions such as finding the absolute value of numbers, and aggregate values by summing values. You can also create your own functions to perform numerical analysis that is unique to your program.

Performing Numerical Operations

Using the OLAP API you perform basic numeric operations using `NumberSource` methods such as `minus`. There are separate versions of each of these methods that you can use to specify a literal `double`, `float`, `int`, or `short` value. There is also a version of each of these method that takes a `NumberSource` as an argument.

The OLAP API methods that you use to perform basic numeric operations include those outlined in Table 6–2.

Table 6–2 OLAP API Methods that Perform Basic Numeric Operations

Method	Description
`div(rhs)`	Creates a new `NumberSource` that has the same structure as the base `NumberSource` but whose values are the values of the base `NumberSource` divided by the specified value.
`intpart()`	Creates a new `NumberSource` that has the same structure as the base `NumberSource` but whose values are the integer portion of the values of the base `NumberSource`.
`minus(rhs)`	Creates a new `NumberSource` that has the same structure as the base `NumberSource` but whose values are the values of the base `NumberSource` minus the specified value.
`negate()`	Creates a new `NumberSource` that has the same structure as the base `NumberSource` but whose values are the values of the base `NumberSource` negated.
`plus(rhs)`	Creates a new `NumberSource` that has the same structure as the base `NumberSource` but whose values are the values of the base `NumberSource` plus the specified value.
`rem(rhs)`	Creates a new `NumberSource` that has the same structure as the base `NumberSource` but whose values are the remainders of the values of the base `NumberSource` when they are divided by the specified value.
`times (rhs)`	Creates a new `NumberSource` that has the same structure as the base `NumberSource` but whose values are the values of the base `NumberSource` multiplied by the specified value.

Subtracting the Same Value From all Values: Example

Assume, as shown below. that there is a `NumberSource` named `unit_Cost` that has outputs of `productsDim` and `timesDim` and a type of `Integer`.

productsDim	timesDim	unit_Cost
Boys	1998	4000
Boys	31-DEC-01	10
49780	1998	500
49780	31-DEC-01	9

Now assume that you want to subtract 10% of the sales from each value of `unit_Cost` to find the adjusted income for each product as shown in which creates a new `Source` named `percentAjustment`.

Example 6–24   Subtracting the Same Value from all Values

```
NumberSource percentAdjustment = unit_Cost.minus(unit_Cost.times(.10));
```

The new `NumberSource`, named `percentAdjustment`, has the following structure and values.

productsDim	timesDim	percentAdjustment
Boys	1998	3600
Boys	31-DEC-01	9
49780	1998	450
49780	...	...
49780	31-DEC-01	8

Subtracting the Values of one NumberSource from Another: Example

Assume that you have the `NumberSource` named `unitCost` described in the previous example and that you also have the `NumberSource` named `unitManufacturingCost` shown below.

productsDim	timesDim	unit_Cost values
Boys	1998	600
Boys	31-DEC-01	3
49780	1998	250
49780	31-DEC-01	2

Now assume that you want to calculate the non-manufacturing for each product. To do this you need to subtract the manufacturing costs from the unit costs. To do this you use the following code which creates a new `Source` named `nonManufacturingCost` by performing the operation on `unit_Cost`.

Example 6–25 Subtracting the Values of one NumberSource from Another

```
NumberSource nonManufacturingCost = unitCost.minus(unitManufacturingCost);
```

`nonManufacturingCost` has the structure and values shown below.

productsDim	timesDim	values
Boys	1998	3400
Boys	31-DEC-01	7
49780	1998	250
49780	31-DEC-01	7

For a more complete explanation of these methods, see OLAP API Javadoc.

Making Numerical Comparisons

The NumberSource class has a number of methods make numerical comparisons. These methods compare each value in a NumberSource to a specified value. These methods return a BooleanSource that has the same structure as the original NumberSource and that has an value that is true when the comparison for a given value of the original NumberSource is true, or false when the comparison is false. There are separate versions of each of these methods that you can use to specify a literal double, float, int, or short value.

The numerical comparison methods provided with the OLAP API include those listed in Table 6–3. For a more complete explanation of these methods, see the OLAP API Javadoc.

Table 6–3 Numerical Comparison Methods

Method	Description
eq	Creates a new BooleanSource, with the same outputs and inputs as the base NumberSource, and with a value of true for each value of the NumberSource that is equal to the specified value and a value of false for each value of the NumberSource that is not.
ge	Creates a new BooleanSource, with the same outputs and inputs as the base NumberSource, and with a value of true for each value of the NumberSource that is greater than or equal to the specified value and a value of false for each value of the NumberSource that is not.
gt	Creates a new BooleanSource, with the same outputs and inputs as the base NumberSource, and with a value of true for each value of the NumberSource that is larger than the specified value and a value of false for each value of the NumberSource that is not.
le	Creates a new BooleanSource, with the same outputs and inputs as the base NumberSource, and with a value of true for each value of the NumberSource that is lesser than or equal to the specified value and a value of false for each value of the NumberSource that is not.
lt	Creates a new BooleanSource, with the same outputs and inputs as the base NumberSource, and with a value of true for each value of the NumberSource that is less than the specified value and a value of false for each value of the NumberSource that is not.
ne	Creates a new BooleanSource, with the same outputs and inputs as the base NumberSource, and with a value of true for each value of the NumberSource that is not equal to the specified value and a value of false for each value of the NumberSource that is equal.

Working with Standard Numerical Functions

The OLAP API has many methods that represent standard numerical functions. These methods include those listed in Table 6–4. You can also write your own functions as described in "Creating Your own Numerical Functions" on page 6-27.

When you use these functions with a `NumberSource`, they return a new `NumberSource` that has the same structure as the original `NumberSource` and whose values are the values of the original `NumberSource` modified according to the function. For example, the `abs()` method returns a new `NumberSource` each of whose values has the absolute value of the corresponding value in the original `NumberSource`.

Table 6–4 Methods that Represent Standard Numerical Functions

Method	Description
`abs()`	Creates a new `NumberSource`, with the same outputs and inputs as the base `NumberSource`, whose values are the absolute value of each value of the base `NumberSource`.
`arccos()`	Creates a new `NumberSource`, with the same outputs and inputs as the base `NumberSource`, whose values are the angle value (in radians) of the value (interpreted as a cosine) of each value of the `NumberSource`.
`arcsin()`	Creates a new `NumberSource`, with the same outputs and inputs as the base `NumberSource`, whose values are the angle value (in radians) of the value (interpreted as a sine) of each value of the `NumberSource`.
`arctan()`	Creates a new `NumberSource`, with the same outputs and inputs as the base `NumberSource`, whose values are the angle value (in radians) of the value (interpreted as a tangent) of each value of the `NumberSource`.
`cos()`	Creates a new `NumberSource`, with the same outputs and inputs as the base `NumberSource`, whose values are the cosine of the value (interpreted as an angle value in radians) of each value of the `NumberSource`.
`cosh()`	Creates a new `NumberSource`, with the same outputs and inputs as the base `NumberSource`, whose values are the hyperbolic cosine of the value (interpreted as an angle value in radians) of each value of the `NumberSource`
`log()`	Creates a new `NumberSource`, with the same outputs and inputs as the base `NumberSource`, whose values are the natural logarithm of each value of the `NumberSource`.

Table 6–4 (Cont.) Methods that Represent Standard Numerical Functions

Method	Description
pow(rhs)	Creates a new NumberSource, with the same outputs and inputs as the base NumberSource, whose values are each value of the NumberSource raised to the specified value.
round(multiple)	Creates a new NumberSource, with the same outputs and inputs as the base NumberSource, whose values are each value of the NumberSource rounded to the nearest multiple of the specified value.
sin()	Creates a new NumberSource, with the same outputs and inputs as the base NumberSource, whose values are the sine of the value (interpreted as an angle) of each value of the NumberSource.

Working with Aggregation Methods

Standard numerical methods like stdev() work on each value in a NumberSource. An aggregation method is a method like total() that uses the values in a series of Source values to perform its calculations. The way that Oracle OLAP processes an aggregation function varies depending on whether or not the base NumberSource has inputs:

- When the base NumberSource does not have any inputs, an aggregation function creates a new NumberSource, without any outputs or inputs, with a single value that is calculated using all of the values in the base NumberSource. (See "Calculating the Sum When a Source Has only Outputs: Example" on page 6-25 for an example.)
- When the base NumberSource has inputs, each set of output values identifies a subset of values (tuples). In this case, an aggregation method works on each subset of data. The aggregation function creates a new NumberSource, without the same outputs as the base NumberSource, with one value for each set of output values. These values are calculated using all of values in the base NumberSource identified by that set of output values. (See "Calculating the Sum When a Source Has only Outputs: Example" on page 6-25 for an example.)

The numerical aggregation methods provided by the OLAP API include the methods in Table 6–5. You can also write your own aggregation functions as described in "Creating Your own Numerical Functions" on page 6-27.

Table 6–5 Aggregation Methods

Method	Description When the NumberSource Does Not Have Inputs
`average`	Creates a new `NumberSource`, without any outputs or inputs, whose value is the average of the values of a `NumberSource`.
`maximum`	Creates a new `NumberSource`, without any outputs or inputs, whose value is the largest value of a `NumberSource`.
`minimum`	Creates a new `NumberSource`, without any output or inputs, whose value is the smallest value of a `NumberSource`.
`total`	Creates a new `NumberSource`, without any outputs or inputs, whose value is the sum of the values of a `NumberSource`.

There are two different versions of each of the numerical aggregation methods. One version excludes all null values when making its calculations. The other version allows you to specify whether or not you want null values included in the calculation.

For more information on how OLAP API methods determine the position of an value and therefore how they determine what values to use when calculating the values of aggregation methods, see Finding the Position of Values on page 6-6.

Calculating the Sum When a Source Has only Outputs: Example

Assume that you have the `Source` named `unitsSoldByCountry` that has two outputs (`products` and `countries`) and whose values are the total number of units for each product sold for each country.

products (output2)	countries (output1)	values of unitsSoldByCountry
395	Australia	1300
395	United States	800
49780	Australia	10050
49780	United States	50

Now assume that you want to total these values. Since both products and countries are outputs, when you issue the code shown below, the new NumberSource calculates the total number of units sold for all products in all countries.

Example 6–26 Calculating the Sum of Values When a Source has only Outputs

```
NumberSource totalUnitsSold = unitsSoldyByCountry.total();
```

The new NumberSource called totalUnitsSold has only a single value that is the total of the values of unitsSoldByCountry.

value of totalUnitsSold
11350

Calculating the Sum When a Source Has an Output and an Input: Example

Assume that you have the Source named unitsSoldByCountry that has an output of countries and an input of products and whose values are the total number of units for each product sold for each country.

countries (output)	values of unitsSoldByCountry
Australia	1300 10050
United States	50 800

Now assume that you total these values. Since product is input, when you issue the code shown below, the new NumberSource calculates the total number of units

sold for all products in each country;. It does not calculate the total for all products in all countries.

Example 6–27 Calculating the Sum of Values When a Source has Outputs and Inputs

```
NumberSource totalUnitsSoldByCountry = unitsSoldByCountry.total();
```

The new `NumberSource` called `totalUnitsSoldByCountry` has an output of `countries` and values shown below.

countries (output)	values of unitsSoldByCountry
Australia	11350
United States	850

Creating Your own Numerical Functions

The alias method can be used to create parameters. Example 6–28, "Creating a Standard Function" shows how to create a new function using the `alias` method. You can only create cell or row calculation functions in this way. To create client aggregation or position-based functions you use the `extract` method.

Creating Your own Standard Function: Example

Example 6–28 creates a function that takes a number and multiplies it by 1.05. The function has one parameter, called `param`, which is created by calling the `alias` method on the fundamental `Source` representing the Number OLAP API data type which is the set of all numbers. (Note how the `value` method is used to make the parameter an input of the function.) The function created in Example 6–28 is effectively the same as the built-in functions provided by the OLAP API. It can be used by joining the function to the parameter and the required parameter expression.

Assume you want to create a product selection defined to be the set of all products for which the `unitsSold` measure is greater than the value specified by a parameter. The parameter must be specified before data can be fetched from this `Source`. You can create this parameter as shown in Example 6–29. To set the value of the parameter to 100, you use the code shown in Example 6–30. You can then

apply the function created in Example 6–28 to a `Source` named `sales` as shown in Example 6–31.

Example 6–28 Creating a Standard Function

```
//Get the Source that represents the number data type
NumberSource number =(NumberSource)dataProvider
   .getFundamentalDefinitionProvider()
   .getNumberDataType()
   .getSource();
//Create a parameter
NumberSource param = (NumberSource)number.alias();
//Create a function
NumberSource function = ((NumberSource)param.value()).times(1.05);
```

Example 6–29 Creating a Parameterized Selection

```
//Get the Source that represents the number data type
NumberSource number = dataProvider
   .getFundamentalDefinitionProvider()
   .getNumberDataType()
   .getSource();
//Create a parameter
NumberSource param = (NumberSource)number.alias();
//Create a parameterized selection
Source products = ...;
NumberSource unitsSold = ...;
Source productSelection = products.select(unitsSold.gt(param.value()));
```

Example 6–30 Setting the Value of the Parameter

```
Source unitsSoldGT100 = productSelection.join(param, 100);
```

Example 6–31 Using a Standard Function You Created

```
//Use the function
NumberSource sales = ...;
NumberSource fsales = function.join(param, sales);
```

Creating Your own Aggregation Function: Example

Assume that you want to create a weighted average function. To do so, you write the code shown in Example 6–32, "Creating a Weighted Average Function". As with the example of a standard function Example 6–28, "Creating a Standard Function", this code first creates a parameter named `param` for the function to use. However, since this is an aggregation function, the code uses the `extract()` method with `param` when it calculates the final result.

To use the weighted average function created in Example 6–32, you issue the code shown in Example 6–33.

Example 6–32 Creating a Weighted Average Function

```
//Define an aggregation function
NumberSource weight = ...;
//Create a parameter
NumberSource param = (NumberSource) number.alias();
//Create a function
NumberSource weightedAverage = param.extract().times(weight).average();
```

Example 6–33 Using the Weighted Average Function Created in Example 6–32

```
//Use the aggregation function
NumberSource sales = ...;
NumberSource paramSales = dp.createConstantSource(param.selectValues(sales));
Source weightedSales = weightedAverage.join(paramSales);
```

Manipulating String Values

The `StringSource` class defines methods that are string-specific versions of various `Source` methods that you can use to append, insert, select, and remove values whose values are Java `String` objects.

The `StringSource` class also has methods listed in Table 6–6 that you can use to manipulate the values of the `StringSource` objects. The OLAP API also provides the methods listed in Table 6–7 that you can use to manipulate substrings within the values of a `StringSource`.

Table 6–6 Methods for Manipulating the values of StringSource Objects

Method	Description
`length()`	Creates a new `NumberSource` with the same structure as the base `StringSource` and whose values are the length of each value of the `StringSource`.
`textFill(width)`	Creates a new `StringSource`, with the same structure as the base `StringSource`, whose values are each value of the base `StringSource` reformatted o the specified width by adding blank spaces.
`toLowercase()`	Creates a new `StringSource`, with the same structure as the base `StringSource`, whose values are the values of the base `StringSource` with all alphabetic characters in lowercase.
`toUppercase()`	Creates a new `StringSource`, with the same structure as the base `StringSource`, whose values are the values of the base `StringSource` with all alphabetic characters in uppercase.
`trim()`	Creates a new `StringSource`, with the same structure as the base `StringSource`, whose values are the values of the base `StringSource` with the leading and trailing blank spaces removed.
`trimLeading()`	Creates a new `StringSource`, with the same structure as the base `StringSource`, whose values are the values of the base `StringSource` with the leading blank spaces removed.
`trimTrailing`	Creates a new `StringSource`, with the same structure as the base `StringSource`, whose values are the values of the base `StringSource` with the trailing blank spaces removed.

Table 6–7 Methods for Manipulating Substrings of StringSource Objects

Method	Description
`indexOf (substring, fromIndex)`	Creates a new `StringSource`, with the same structure as the base `StringSource`, whose values are the specified substrings in the values of the base `StringSource` that begin after the specified character position.
`remove (index, length)`	Creates a new `StringSource`, with the same structure as the base `StringSource`, whose values a re the values of the base `StringSource` with the characters between the specified character positions removed.
`replace (oldString, newString)`	Creates a new `StringSource`, with the same structure as the base `StringSource`, whose values are the values of the base `StringSource` with the specified substring, and replaced with a different substring.
`substring (index, length)`	Creates a new `StringSource`, with the same structure as the base `StringSource`, whose values are the specified substrings in the values of the base `StringSource`.

There are two different versions of each of these methods. In one version you specify the values using `Source` objects, in the other you specify the values using literal values.

7 Using a TransactionProvider

This chapter describes the Oracle OLAP API `Transaction` and `TransactionProvider` interfaces and describes how you use implementations of those interfaces in an application. You must create a `TransactionProvider` before you can create a `DataProvider`, and you must use methods on the `TransactionProvider` to prepare and commit a `Transaction` before you can create a `Cursor` for a derived `Source`.

This chapter includes the following topics:

- About Creating a Query in a Transaction
- Using TransactionProvider Objects

About Creating a Query in a Transaction

The Oracle OLAP API is transactional. Each step in creating a query occurs in the context of a `Transaction`. One of the first actions of an OLAP API application is to create a `TransactionProvider`. The `TransactionProvider` provides `Transaction` objects to the application.

The `TransactionProvider` ensures the following:

- A `Transaction` is isolated from other `Transaction` objects. Operations performed in a `Transaction` are not visible in, and do not affect, other `Transaction` objects.
- If an operation in a `Transaction` fails, its effects are undone (the `Transaction` is rolled back).
- The effects of a completed `Transaction` persist.

When you create a derived `Source` by calling a method on another `Source`, that `Source` is created in the context of the *current* `Transaction`. The `Source` is *active* in the `Transaction` in which you create it or in a child `Transaction` of that `Transaction`.

You get or set the current `Transaction`, or begin a child `Transaction`, by calling methods on a `TransactionProvider`. In a child `Transaction` you can change the state of a `Template` that you created in the parent `Transaction`. By displaying the data specified by the `Source` produced by the `Template` in the parent `Transaction` and also displaying the data specified by the `Source` produced by the `Template` in the child `Transaction`, you can provide the end user of your application with the means of performing what-if analysis.

Types of Transaction Objects

The OLAP API has the following two types of Transaction objects:

- A *read* Transaction. Initially, the current Transaction is a read Transaction. A read Transaction is required for creating a Cursor to fetch data from Oracle OLAP. For more information on Cursor objects, see Chapter 9.
- A *write* Transaction. A write Transaction is required for creating a derived Source or for changing the state of a Template. For more information on creating a derived Source, see Chapter 5. For information on Template objects, see Chapter 10.

In the initial read Transaction, if you create a derived Source or if you change the state of a Template object, then a child write Transaction is automatically generated. That child Transaction becomes the current Transaction.

If you then create another derived Source or change the Template state again, that operation occurs in the same write Transaction. You can create any number of derived Source objects, or make any number of Template state changes, in that same write Transaction. You can use those Source objects, or the Source produced by the Template, to define a complex query.

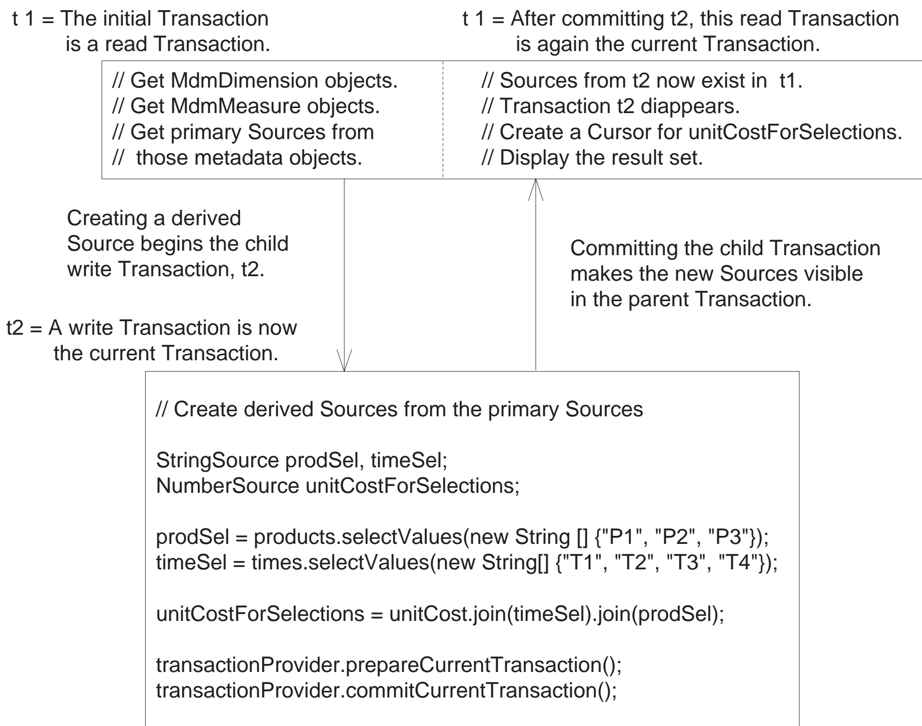
Before you can create a Cursor to fetch the result set specified by a derived Source, you must move the Source from the child write Transaction into the parent read Transaction. To do so, you prepare and commit the Transaction.

Preparing and Committing a Transaction

To move a Source that you created in a child Transaction into the parent read Transaction, call the prepareCurrentTransaction and commitCurrentTransaction methods on the TransactionProvider. When you commit a child write Transaction, a Source you created in the child Transaction moves into the parent read Transaction. The child Transaction disappears and the parent Transaction becomes the current Transaction. The Source is active in the current read Transaction and you can therefore create a Cursor for it.

The following figure illustrates the process of moving a `Source` created in a child write `Transaction` into its parent read `Transaction`.

Figure 7–1 Committing a Write Transaction into Its Parent Read Transaction



About Transaction and Template Objects

Getting and setting the current `Transaction`, beginning a child `Transaction`, and rolling back a `Transaction` are operations that you use to allow an end user to make different selections starting from a given state of a dynamic query. This creating of alternatives based on an initial state is known as what-if analysis.

To present the end user with alternatives based on the same initial query, you do the following:

1. Create a `Template` in a parent `Transaction` and set the initial state for the `Template`.
2. Get the `Source` produced by the `Template`, create a `Cursor` to retrieve the result set, get the values from the `Cursor`, and then display the results to the end user.
3. Begin a child `Transaction` and modify the state of the `Template`.
4. Get the `Source` produced by the `Template` in the child `Transaction`, create a `Cursor`, get the values, and display them.

You can then replace the first `Template` state with the second one or discard the second one and retain the first.

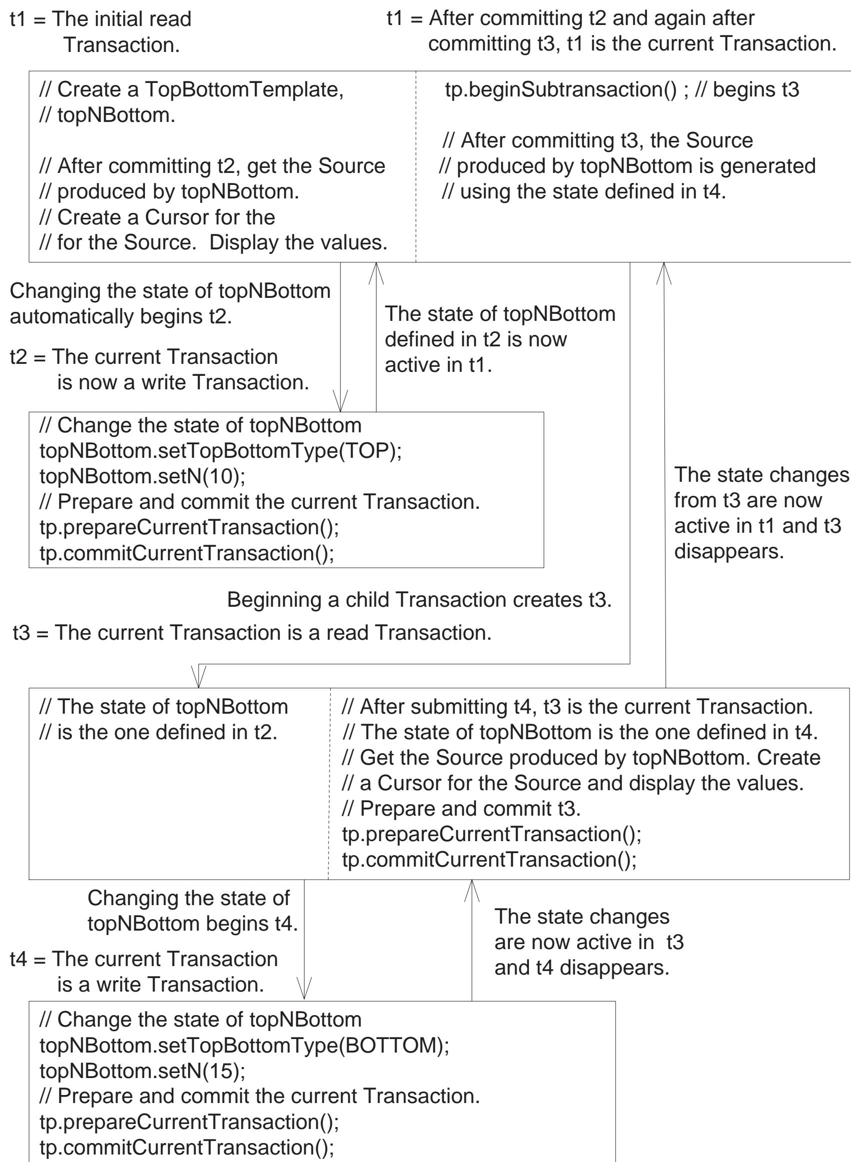
Beginning a Child Transaction

To begin a child read `Transaction`, call the `beginSubtransaction` method on the `TransactionProvider` you are using. If you then change the state of a `Template`, a child write `Transaction` begins automatically. The write `Transaction` is a child of the child read `Transaction`.

To get the data specified by the `Source` produced by the `Template`, you prepare and commit the write `Transaction` into its parent read `Transaction`. You can then create a `Cursor` to fetch the data. The changed state of the `Template` is not visible in the original parent. The changed state does not become visible in the parent until you prepare and commit the child read `Transaction` into the parent read `Transaction`.

The following figure illustrates beginning a child read `Transaction`, creating `Source` objects in a write `Transaction`,and committing the write `Transaction` into its parent read `Transaction`. The figure then shows committing the child read `Transaction` into its parent read `Transaction`. In the figure, `tp` is the `TransactionProvider`.

Figure 7–2 Committing a Child Read Transaction into Its Parent Transaction



After beginning a child read `Transaction`, you can begin a child read `Transaction` of that child, or a grandchild of the initial parent `Transaction`. For an example of creating child and grandchild `Transaction` objects, see Example 7–2.

About Rolling Back a Transaction

You roll back, or undo, a `Transaction` by calling the `rollbackCurrentTransaction` method on the `TransactionProvider` you are using. Rolling back a `Transaction` discards any changes that you made during that `Transaction` and makes the `Transaction` disappear.

Before rolling back a `Transaction`, you must close any `CursorManager` objects you created in that `Transaction`. After rolling back a `Transaction`, any `Source` objects that you created or `Template` state changes that you made in the `Transaction` are no longer valid. Any `Cursor` objects you created for those `Source` objects are also invalid.

Once you roll back a `Transaction`, you cannot prepare and commit that `Transaction`. Likewise, once you commit a `Transaction`, you cannot roll it back.

Example 7–1 Rolling Back a Transaction

The following example creates a `TopBottomTemplate` and sets its state. The example begins a child `Transaction` that sets a different state for the `TopBottomTemplate` and then rolls back the child `Transaction`. The `TransactionProvider` is `tp`.

```
// The current Transaction is a read Transaction, t1.
// Create a TopBottomTemplate using product as the base
// and dp as the DataProvider.
TopBottomTemplate topNBottom = new TopBottomTemplate(product, dp);

// Changing the state of a Template requires a write Transaction, so a
// write child Transaction, t2, is automatically started.
topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_TOP);
topNBottom.setN(10);
topNBottom.setCriterion(singleSelections.getSource());

// Prepare and commit the Transaction t2.
tp.prepareCurrentTransaction();
tp.commitCurrentTransaction();           //t2 disappears
```

```
// The current Transaction is now t1.
// Create a Cursor and display the results (operations not shown).

// Start a child Transaction, t3. It is a read Transaction.
tp.beginSubtransaction();          // t3 is the current Transaction

// Change the state of topNBottom. Changing the state requires a
// write Transaction so Transaction t4 starts automatically,
topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_BOTTOM);
topNBottom.setN(15);

// Prepare and commit the Transaction.
tp.prepareCurrentTransaction();
tp.commitCurrentTransaction();           // t4 disappears

// Create a Cursor and display the results. // t3 is the current Transaction
// Close the CursorManager for the Cursor created in t3.
// Undo t3, which discards the state of topNBottom that was set in t4.
tp.rollbackCurrentTransaction()         // t3 disappears

// Transaction t1 is now the current Transaction and the state of
// topNBottom is the one defined in t2.
```

Getting and Setting the Current Transaction

You get the current `Transaction` by calling the `getCurrentTransaction` method on the `TransactionProvider` you are using, as in the following example.

```
Transaction t1 = getCurrentTransaction();
```

To make a previously saved `Transaction` the current `Transaction`, you call the `setCurrentTransaction` method on the `TransactionProvider`, as in the following example.

```
setCurrentTransaction(t1);
```

Using TransactionProvider Objects

In the Oracle OLAP API, the `TransactionProvider` interface is implemented by the `ExpressTransactionProvider` concrete class. Before you create a `DataProvider`, you must create a new instance of an `ExpressTransactionProvider`. You then pass that `TransactionProvider` to the `DataProvider` constructor. The `TransactionProvider` provides `Transaction` objects to your application.

As described in "Preparing and Committing a Transaction" on page 7-3, you use the prepareCurrentTransaction and commitCurrentTransaction methods to make a derived Source that you created in a child write Transaction visible in the parent read Transaction. You can then create a Cursor for that Source.

If you are using Template objects in your application, you might also use the other methods on TransactionProvider to do the following:

- Begin a child Transaction.
- Get the current Transaction so you can save it.
- Set the current Transaction to a previously saved one.
- Rollback, or undo, the current Transaction, which discards any changes made in the Transaction. Once a Transaction has been rolled back, it is invalid and cannot be committed. Once a Transaction has been committed, it cannot be rolled back. If you created a Cursor for a Source in a Transaction, you must close the CursorManager before rolling back the Transaction.

To demonstrate how to use Transaction objects to modify dynamic queries, Example 7–2 builds on the TopBottomTest application defined in Chapter 10. To help track the Transaction objects, the example saves the different Transaction objects with calls to the getCurrentTransaction method.

Replace the last five lines of the code from the TopBottomTest class with the code from Example 7–2.

Example 7–2 Using Child Transaction Objects

```
// The parent Transaction is the current Transaction at this point.
// Save the parent read Transaction as parentT1.
Transaction parentT1 = tp.getCurrentTransaction();

// Begin a child Transaction of parentT1.
tp.beginSubtransaction();  // This is a read Transaction.

// Save the child read Transaction as childT2.
Transaction childT2 = tp.getCurrentTransaction();

// Change the state of the TopBottomTemplate. This starts a
// write Transaction, a child of the read Transaction childT2.
topNBottom.setN(15);
topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_BOTTOM);
```

```
// Save the child write Transaction as writeT3.
Transaction writeT3 = tp.getCurrentTransaction();

// Prepare and commit the write Transaction writeT3.
try{
  context.getTransactionProvider().prepareCurrentTransaction();
}
catch(NotCommittableException e){
 System.out.println("Caught exception " + e + ".");
}
context.getTransactionProvider().commitCurrentTransaction();

// The commit moves the changes made in writeT3 into its parent,
// the read Transaction childT2. The writeT3 Transaction
// disappears. The current Transaction is now childT2
// again but the state of the TopBottomTemplate has changed.

// Create a Cursor and display the results of the changes to the
 // TopBottomTemplate that are visible in childT2.
createCursor(topNBottom.getSource());

// Begin a grandchild Transaction of the initial parent.
tp.beginSubtransaction();  // This is a read Transaction.

// Save the grandchild read Transaction as grandchildT4.
Transaction grandchildT4 = tp.getCurrentTransaction();

// Change the state of the TopBottomTemplate. This starts another
// write Transaction, a child of grandchildT4.
topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_TOP);

// Save the write Transaction as writeT5.
Transaction writeT5 = tp.getCurrentTransaction();

// Prepare and commit writeT5.
try{
  context.getTransactionProvider().prepareCurrentTransaction();
}
catch(NotCommittableException e){
  System.out.println("Caught exception " + e + ".");
}
context.getTransactionProvider().commitCurrentTransaction();
```

```
    // Transaction grandchildT4 is now the current Transaction and the
    // changes made to the TopBottomTemplate state are visible.

    // Create a Cursor and display the results visible in grandchildT4.
    createCursor(topNBottom.getSource());

    // Commit the grandchild into the child.
    try{
      context.getTransactionProvider().prepareCurrentTransaction();
    }
    catch(NotCommittableException e){
      System.out.println("Caught exception " + e + ".");
    }
    context.getTransactionProvider().commitCurrentTransaction();

    // Transaction childT2 is now the current Transaction.
    // Instead of preparing and committing the grandchild Transaction,
    // you could rollback the Transaction, as in the following
    // method call:
    // rollbackCurrentTransaction();
    // If you roll back the grandchild Transaction, then the changes
    // you made to the TopBottomTemplate state in the grandchild
    // are discarded and childT2 is the current Transaction.

    // Commit the child into the parent.
    try{
      context.getTransactionProvider().prepareCurrentTransaction();
    }
    catch(NotCommittableException e){
      System.out.println("Caught exception " + e + ".");
    }
    context.getTransactionProvider().commitCurrentTransaction();

    // Transaction parentT1 is now the current Transaction. Again,
    // you could roll back the childT2 Transaction instead of
    // preparing and committing it. If you did so, then the changes
    // you made in childT2 are discarded. The current Transaction
    // would be parentT1, which would have the original state of
    // the TopBottomTemplate, without any of the changes made in
    // the grandchild or the child transactions.

  }  // end of main() method
}  // end of TopBottomTest class
```

8

Understanding Cursor Classes and Concepts

This chapter describes the Oracle OLAP API `Cursor` class and its related classes, which you use to retrieve and gain access to the results of a query. This chapter also describes the `Cursor` concepts of position, fetch size, and extent. For examples of creating and using a `Cursor` and its related objects, see Chapter 9.

This chapter includes the following topics:

- Overview of the OLAP API Cursor Objects
- About Cursor Positions and Extent
- About Fetch Sizes and Fetch Blocks

Overview of the OLAP API Cursor Objects

A Cursor retrieves the result set defined by a Source. Creating a Cursor for a Source requires at least two intermediate steps. After creating a Source that defines the data that you want to retrieve from the data store, you create a Cursor for that Source by doing the following:

1. Creating a CursorManagerSpecification by passing the Source to the createCursorManagerSpecification method on the DataProvider that you are using. The CursorManagerSpecification has CursorSpecification objects in a structure that mirrors the structure of the Source.
2. Creating a CursorManager by calling the createCursorManager method on the DataProvider and passing it the CursorManagerSpecification. The CursorManager creates Cursor objects. It also manages the local data cache for its Cursor objects and is aware of changes to the Source for a dynamic query. If the Source for the CursorManagerSpecification has inputs, then you must also pass to the createCursorManager method an array of Source objects for those inputs.
3. Creating a Cursor by calling the createCursor method on the CursorManager. The structure of the Cursor mirrors the structures of the CursorManagerSpecification and the Source. The CursorSpecification objects of a CursorManagerSpecification specify the behavior of their corresponding Cursor objects. If the Source for the CursorManagerSpecification has inputs, then you must also pass to the createCursor method an array of CursorInput objects that specify values for the input Source objects.

For an example of creating a Cursor, see Chapter 9.

This architecture provides great flexibility in fetching data from a result set and in selecting data to display. You can do the following:

- Create more than one CursorManagerSpecification object for the same Source. You can specify different behavior on the CursorSpecification components of the various CursorManagerSpecification objects in order to retrieve and display different sets of values from the same result set. You might want to do this when displaying the data from a Source in different formats, such as in a table and a crosstab.
- Receive notification that the Source produced by the Template has changed. If you add a CursorManagerUpdateListener to the CursorManager for a Source, then the CursorManager notifies the

CursorManagerUpdateListener when the Source for a dynamic query has changed and you that therefore need to update the CursorManagerSpecification for the CursorManager.

- Update the CursorManagerSpecification for a CursorManager. If you are using Template objects to produce a dynamic query and the state of a Template changes, then the Source produced by the Template changes. If you have created a Cursor for the Source produced by the Template, then you need to replace the CursorManagerSpecification for the CursorManager with an updated CursorManagerSpecification for the changed Source. You can then create a new Cursor from the CursorManager.
- Create different of Cursor objects from the same CursorManager and set different fetch sizes on those Cursor objects. You might do this when you want to display the same data as a table and as a graph.

Sources For Which You Cannot Create a Cursor

Some Source objects do not specify data that a Cursor can retrieve from the data store. The following are Source objects for which you cannot create a Cursor.

- A Source that specifies an operation that is not computationally possible. An example is a Source that specifies an infinite recursion.
- A Source that defines an infinite result set. An example is the fundamental Source that represents the set of all String objects.
- A Source that has no elements or includes another Source that has no elements. Examples are a Source returned by the getEmptySource method on DataProvider and another Source derived from the empty Source. Another example is a derived Source that results from selecting a value from a primary Source that you got from an MdmDimension and the selected value does not exist in the dimension.

Cursor Objects and Transaction Objects

When you create a derived Source or change the state of a Template, you create the Source in the context of the current Transaction. The Source is *active* in the Transaction in which you create it or in a child Transaction of that Transaction. A Source must be active in the current Transaction for you to be able to create a Cursor for it.

Creating a derived Source occurs in a *write* Transaction. Creating a Cursor occurs in a *read* Transaction. After creating a derived Source, and before you can create a Cursor for that Source, you must change the write Transaction into a read Transaction by calling the prepareCurrentTransaction and commitCurrentTransaction methods on the TransactionProvider your application is using. For information on Transaction and TransactionProvider objects, see Chapter 7.

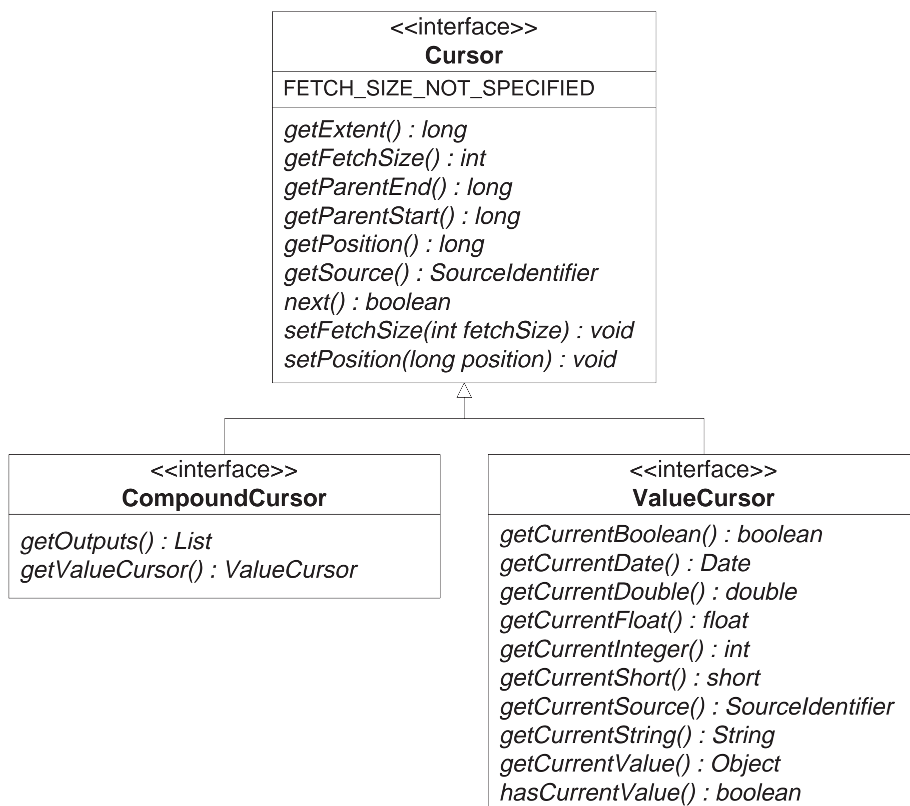
Cursor Class

In the oracle.olapi.data.cursor package, the Oracle OLAP API defines the interfaces described in the following table.

Interface	Description
Cursor	An abstract superclass that encapsulates the notion of a *current position.*
ValueCursor	A Cursor that has a value at the current position. A ValueCursor has no child Cursor objects.
CompoundCursor	Cursor that has child Cursor objects, which are a child ValueCursor for the values of its Source and an output child Cursor for each output of the Source.

Figure 8–1shows the class hierarchy of the Cursor classes. The CompoundCursor and ValueCursor interfaces extend the Cursor interface.

Figure 8–1 Cursor Hierarchy



Structure of a Cursor

The structure of a `Cursor` mirrors the structure of its `Source`. If the `Source` does not have any outputs, the `Cursor` for that `Source` is a `ValueCursor`. If the `Source` has one or more outputs, the `Cursor` for that `Source` is a `CompoundCursor`. A `CompoundCursor` has as children a base `ValueCursor`, which has the values of the base of the `Source` of the `CompoundCursor`, and one or more output `Cursor` objects.

The output of a `Source` is another `Source`. An output `Source` can itself have outputs. The child `Cursor` for an output of a `Source` is a `ValueCursor` if the output `Source` does not have any outputs and a `CompoundCursor` if it does.

For example, suppose you have created a derived `Source` called `productSel` that represents a selection of product identification values from a primary `Source` that represents values from a dimension of products. You have selected `815`, `1050`, and `2055` as the values for `productSel`. If you create a `Cursor` for `productSel`, then that `Cursor` is a `ValueCursor` because `productSel` has no outputs.

You have also created a derived `Source` called `timeSel` that represents a selection of day values from a primary `Source` that represents a dimension of time values. The values of `timeSel` are `1-JAN-00`, `1-APR-00`, `1-JUL-00`, and `1-OCT-00`.

You have an `MdmMeasure` that represents values for the price of product units. The `MdmMeasure` has as inputs the `MdmDimension` objects representing products and times. You get a `Source` called `unitPrice` from the measure. The `Source` has products and times as inputs.

You join `productSel` and `timeSel` to `unitPrice` to create a `Source`, `unitPriceByDay`, which has `productSel` and `timeSel` as outputs, as in the following:

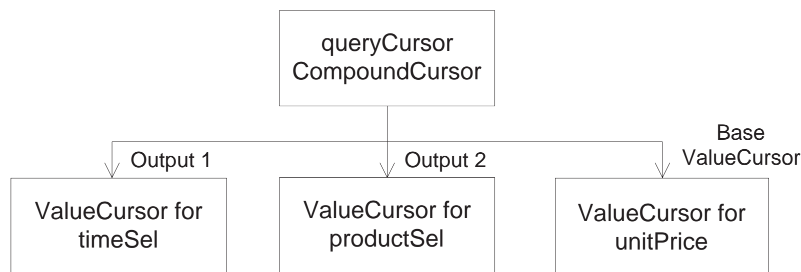
```
unitPriceByDay = unitPrice.join(productSel).join(timeSel);
```

The result set defined by `unitPriceByDay` is unit price values organized by the outputs. Since `timeSel` is joined to the result of `unitPrice.join(productSel)`, `timeSel` is the slower varying output, which means that the result set specifies the set of selected products for each selected time value. For each time value the result set has three product values so the product values vary faster than the time values. The values of the base `ValueCursor` of `unitPriceByDay` are the fastest varying of all, because there is one price value for each product for each day.

You then create a `Cursor`, `queryCursor`, for `unitPriceByDay`. Since `unitPriceByDay` has outputs, `queryCursor` is a `CompoundCursor`. The base `ValueCursor` of `queryCursor` has values from `unitPrice`, which is the base `Source` of the operation that created `unitPriceByDay`. The outputs for `queryCursor` are a `ValueCursor` that has values from `productSel` and a `ValueCursor` that has values from `timeSel`.

Figure 8–2 illustrates the structure of queryCursor. The base ValueCursor and the two output ValueCursor objects are the children of queryCursor, which is the parent CompoundCursor.

Figure 8–2 Structure of the queryCursor CompoundCursor



The following table displays the values from queryCursor in a table. The left column has time values, the middle column has product values, and the right column has the unit price of the specified product on the specified day.

Day	Product	Price of Unit
01-JAN-00	815	58
01-JAN-00	1050	24
01-JAN-00	2055	24
01-APR-00	815	59
01-APR-00	1050	24
01-APR-00	2055	25
01-JUL-00	815	59
01-JUL-00	1050	25
01-JUL-00	2055	25
01-OCT-00	815	61
01-OCT-00	1050	25
01-OCT-00	2055	26

For examples of getting the values from a ValueCursor, see Chapter 9.

Specifying the Behavior of a Cursor

The CursorSpecification objects of a CursorManagerSpecification specify some aspects of the behavior of their corresponding Cursor objects. You must specify the behavior on a CursorSpecification before creating the corresponding Cursor. To specify the behavior, use the following CursorSpecification methods:

- setDefaultFetchSize
- setExtentCalculationSpecified
- setParentEndCalculationSpecified
- setParentStartCalculationSpecified
- specifyDefaultFetchSizeOnChildren
  (for a CompoundCursorSpecification only)

A CursorSpecification also has methods that you can use to discover if the behavior is specified. Those methods are the following:

- isExtentCalculationSpecified
- isParentEndCalculationSpecified
- isParentStartCalculationSpecified

If you have used the CursorSpecification methods to set the default fetch size, or to calculate the extent or the starting or ending positions of a value in its parent, you can successfully use the following Cursor methods:

- getExtent
- getFetchSize
- getParentEnd
- getParentStart
- setFetchSize

For examples of specifying Cursor behavior, see Chapter 9. For information on fetch sizes, see "About Fetch Sizes and Fetch Blocks" on page 8-27. For information on the extent of a Cursor, see"What is the Extent of a Cursor?" on page 8-25. For information on the starting and ending positions in a parent Cursor of the current value of a Cursor, see "About the Parent Starting and Ending Positions in a Cursor" on page 8-22.

CursorManagerSpecification Class

A CursorManagerSpecification for a Source has one or more CursorSpecification objects. The structure of those objects reflects the structure of the Source. For example, a Source that has outputs has a top-level, or *root*, CursorSpecification for the Source, a child CursorSpecification for the values of the Source, and a child CursorSpecification for each output of the Source.

A Source that does not have any outputs has only one set of values. A CursorManagerSpecification for that Source therefore has only one CursorSpecification. That CursorSpecification is the root CursorSpecification of the CursorManagerSpecification.

You can create a CursorManagerSpecification for a multidimensional Source that has one or more inputs. If you do so, then you need to supply a Source for each input when you create a CursorManager for the CursorManagerSpecification. You must also supply a CursorInput for each input Source when you create a Cursor from the CursorManager. You might create a CursorManagerSpecification for a Source with inputs if you want to use a CursorManager to create a series of Cursor objects with each Cursor retrieving data specified by a different set of single values for the input Source objects.

The structure of a Cursor reflects the structure of its CursorManagerSpecification. A Cursor can be a single ValueCursor, for a Source with no outputs, or a CompoundCursor with child Cursor objects, for a Source with outputs. Each Cursor corresponds to a CursorSpecification in the CursorManagerSpecification. You use CursorSpecification methods to specify aspects of the behavior of the corresponding Cursor.

If your application uses Template objects, and a change occurs in the state of a Template so that the structure of the Source produced by the Template changes, then any CursorManagerSpecification objects that the application created for the Source expire. If a CursorManagerSpecification expires, you must create a new CursorManagerSpecification. You can then either use the new CursorManagerSpecification to replace the old CursorManagerSpecification of a CursorManager or use it to create a new CursorManager. You can discover if a CursorManagerSpecification has expired by calling the isExpired method on the CursorManagerSpecification.

CursorSpecification Class

A CursorSpecification specifies certain aspects of the behavior of the Cursor that corresponds to it. You do not create a CursorSpecification directly. You pass a Source to the createCursorManagerSpecification method of a DataProvider and the CursorManagerSpecification returned has a root CursorSpecification for that Source. If the Source has outputs, the CursorManagerSpecification also has a child CursorSpecification for the values of the Source and one for each output of the Source.

With CursorSpecification methods, you can do the following:

- Get the Source that corresponds to the CursorSpecification.
- Get or set the default fetch size for the corresponding Cursor.
- On a CompoundCursorSpecification, specify that the default fetch size is set on the children of the corresponding Cursor.
- Specify that Oracle OLAP should calculate the extent of a Cursor.
- Determine if calculating the extent is specified.
- Specify that Oracle OLAP should calculate the starting or ending position of the current value of the corresponding Cursor in its parent Cursor. If you know the starting and ending positions of a value in the parent, then you can determine how many faster varying elements the parent Cursor has for that value.
- Determine if calculating the starting or ending position of the current value of the corresponding Cursor in its parent is specified.
- Accept a CursorSpecificationVisitor.

For more information, see "About Cursor Positions and Extent" on page 8-16 and "About Fetch Sizes and Fetch Blocks" on page 8-27.

In the oracle.olapi.data.source package, the Oracle OLAP API defines the classes described in the following table.

Class	Description
CursorSpecification	An abstract superclass that implements methods inherited by its subclasses.
ValueCursorSpecification	A CursorSpecification for a Source that has values and no outputs.
CompoundCursorSpecification	A CursorSpecification for a Source that has one or more outputs. A CompoundCursorSpecification has component child CursorSpecification objects.

A Cursor has the same structure as its CursorManagerSpecification. For every ValueCursorSpecification or CompoundCursorSpecification of a CursorManagerSpecification, a Cursor has a corresponding ValueCursor or CompoundCursor. To be able to get certain information or behavior from a Cursor, your application must specify that it wants that information or behavior by calling methods on the corresponding CursorSpecification before it creates the Cursor.

CursorInput Class

A CursorInput provides a value for a Source that you include in the array of Source objects that is the inputSources argument to the createCursorManager method on a DataProvider. If you create a CursorManagerSpecification for a Source that has one or more inputs, then you must provide an inputSources argument when you create a CursorManager for that CursorManagerSpecification. You include a Source in the inputSources array for each input of the Source that you pass to the createCursorManagerSpecification method.

When you create a CursorInput object, you can specify either a single value or a ValueCursor. If you specify a ValueCursor, you can call the synchronize method on the CursorInput to make the value of the CursorInput be the current value of the ValueCursor.

CursorManager Class

A CursorManager manages the buffering of data for the Cursor objects it creates. To create a CursorManager, call the createCursorManager method on a DataProvider and pass it a CursorManagerSpecification. If the Source for the CursorManagerSpecification has one or more inputs, then also pass an array of Source objects to the createCursorManager method. Include in the array a Source for each input.

You can create more than one Cursor from the same CursorManager, which is useful for displaying data from a result set in different formats such as a table or a graph. All of the Cursor objects created by a CursorManager have the same specifications, such as the default fetch sizes and the levels at which fetch sizes are set. Because the Cursor objects have the same specifications, they can share the data managed by the CursorManager.

A CursorManager has methods for creating a Cursor, for discovering whether the CursorManagerSpecification for the CursorManager needs updating, and for adding or removing a CursorManagerUpdateListener. The SpecifiedCursorManager interface adds methods for updating the CursorManagerSpecification, for discovering if the SpecifiedCursorManager is open, and for closing it. The createCursorManager method on DataProvider returns an implementation of the SpecifiedCursorManager interface.

When your application no longer needs a SpecifiedCursorManager, it should close it to free resources in the application and in Oracle OLAP. To close the SpecifiedCursorManager, call its close method.

Updating the CursorManagerSpecification for a CursorManager

If your application is using OLAP API Template objects and the state of a Template changes in a way that alters the structure of the Source produced by the Template, then any CursorManagerSpecification objects for the Source are no longer valid. You need to create new CursorManagerSpecification objects for the changed Source.

After creating a new CursorManagerSpecification, you can create a new CursorManager for the Source. You do not, however, need to create a new CursorManager. You can call the updateSpecification method on the existing CursorManager to replace the previous CursorManagerSpecification with the new CursorManagerSpecification. You can then create a new Cursor from the CursorManager.

To determine if the CursorManagerSpecification for a CursorManager needs updating, call the isSpecificationUpdateNeeded method on the CursorManager. You can also use a CursorManagerUpdateListener to listen for events generated by changes in a Source. For more information, see "CursorManagerUpdateListener Class" on page 8-15.

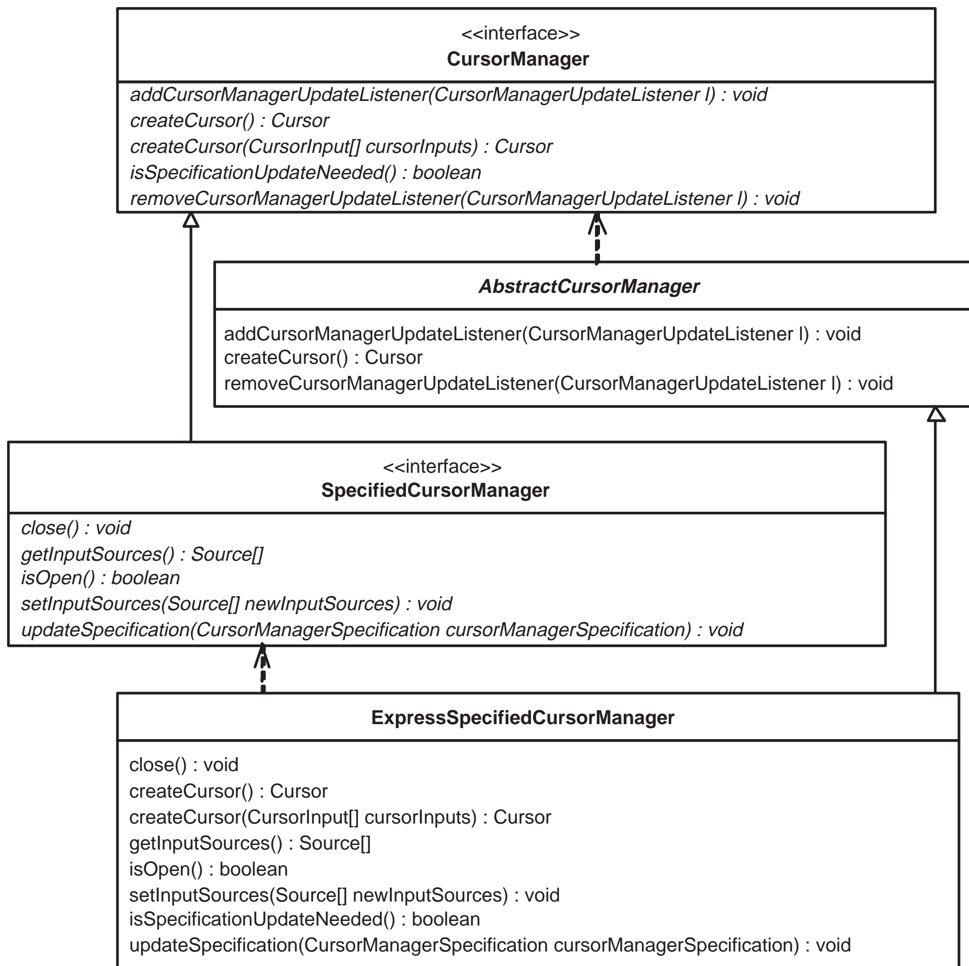
CursorManager Class Hierarchy

The following table lists most of the CursorManager interfaces and classes.

Interface or Class	Description
CursorManager	An interface that has defines methods for all CursorManager objects.
AbstractCursorManager	A CursorManager that implements methods for adding and removing CursorManagerUpdateListener objects. For more information, see "CursorManagerUpdateListener Class" on page 8-15.
SpecifiedCursorManager	An interface that defines additional methods for a CursorManager.
ExpressSpecifiedCursorManager	A class that implements the SpecifiedCursorManager interface and extends AbstractCursorManager. In the Oracle OLAP API, the createCursorManager method on DataProvider returns an instance of this class.

Figure 8–3 shows the relationships of the `CursorManager` classes described in the preceding table. A solid line and a closed arrowhead indicate that a class extends the class to which the arrow points. A dotted line and an open arrowhead indicate that the class implements the interface to which the arrow points.

Figure 8–3 CursorManager Hierarchy



CursorManagerUpdateListener Class

CursorManagerUpdateListener is an interface that has methods that receive CursorManagerUpdateEvent objects. Oracle OLAP generates a CursorManagerUpdateEvent object in response to a change that occurs in a Source that is produced by a Template or when a CursorManager updates its CursorManagerSpecification. Your application can use a CursorManagerUpdateListener to listen for events that indicate it might need to create new Cursor objects from the CursorManager or to update its display of data from a Cursor.

To use a CursorManagerUpdateListener, implement the interface, create an instance of the class, and then add the CursorManagerUpdateListener to the CursorManager for a Source. When a change to the Source occurs, the CursorManager calls the appropriate method on the CursorManagerUpdateListener and passes it a CursorManagerUpdateEvent. Your application can then perform the tasks needed to generate new Cursor objects and update the display of values from the result set that the Source defines.

You can implement more than one version of the CursorManagerUpdateListener interface. You can add instances of them to the same CursorManager.

CursorManagerUpdateEvent Class

Oracle OLAP generates a CursorManagerUpdateEvent object in response to a change that occurs in a Source that is produced by a Template or when a CursorManager updates its CursorManagerSpecification.

You do not directly create instances of this class. Oracle OLAP generates CursorManagerUpdateEvent objects and passes them to the appropriate methods of any CursorManagerUpdateListener objects you have added to a CursorManager. The CursorManagerUpdateEvent has a field that indicates the type of event that occurred. A CursorManagerUpdateEvent has methods you can use to get information about it.

About Cursor Positions and Extent

A Cursor has one or more positions. The ***current position*** of a Cursor is the position that is currently active in the Cursor. To move the current position of a Cursor call the setPosition or next methods on the Cursor.

Oracle OLAP does not validate the position that you set on the Cursor until you attempt an operation on the Cursor, such as calling the getCurrentValue method. If you set the current position to a negative value or to a value that is greater than the number of positions in the Cursor and then attempt a Cursor operation, the Cursor throws a PositionOutOfBoundsException.

The extent of a Cursor is described in "What is the Extent of a Cursor?" on page 8-25.

Positions of a ValueCursor

The current position of a ValueCursor specifies a value, which you can retrieve. For example, productSel, a derived Source described in "Structure of a Cursor" on page 8-5, is a selection of three products from a primary Source that specifies a dimension of products and their hierarchical groupings. The ValueCursor for productSel has three elements. The following example gets the position of each element of the ValueCursor, and displays the value at that position. The output object is a PrintWriter.

```
// productSelValCursor is the ValueCursor for productSel
do {
  output.print(productSelValCursor.getPosition + " : ");
  output.println(productSelValCursor.getCurrentValue);
}
while(productSelValCursor.next());
```

The preceding example displays the following:

```
1 : 815
2 : 1050
3 : 2055
```

The following example sets the current position of `productSelValCursor` to 2 and retrieves the value at that position.

```
productSelValCursor.setPosition(2);
output.println(productSelValCursor.getCurrentValue);
```

The preceding example displays the following:

```
1050
```

For more examples of getting the current value of a `ValueCursor`, see Chapter 9.

Positions of a CompoundCursor

A `CompoundCursor` has one position for each set of the elements of its descendent `ValueCursor` objects. The current position of the `CompoundCursor` specifies one of those sets.

For example, `unitPriceByDay`, the `Source` described in "Structure of a Cursor" on page 8-5, has values from a measure, `unitPrice`. The values are the prices of product units at different times. The outputs of `unitPriceByDay` are `Source` objects that represent selections of four day values from a time dimension and three product values from a product dimension.

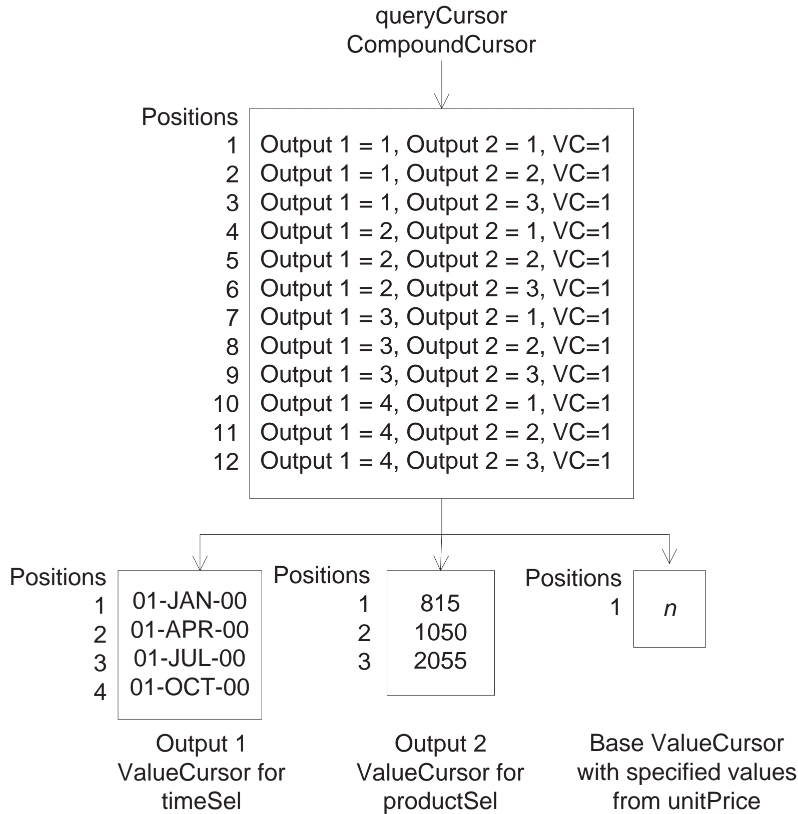
The result set for `unitPriceByDay` has one measure value for each *tuple* (each set of output values), so the total number of values is twelve (one value for each of the three products for each of the four days). Therefore, the `queryCursor` `CompoundCursor` created for `unitPriceByDay` has twelve positions.

Each position of `queryCursor` specifies one set of positions of its outputs and its base `ValueCursor`. For example, position 1 of `queryCursor` defines the following set of positions for its outputs and its base `ValueCursor`:

- Position 1 of output 1 (the `ValueCursor` for `timeSel`)
- Position 1 of output 2 (the `ValueCursor` for `productSel`)
- Position 1 of the base `ValueCursor` for `queryCursor` (This position has the value from the `unitPrice` measure that is specified by the values of the outputs.)

Figure 8–4 illustrates the positions of `queryCursor` `CompoundCursor`, its base `ValueCursor`, and its outputs.

Figure 8–4 Cursor Positions in queryCursor



The `ValueCursor` for `queryCursor` has only one position because only one value of `unitPrice` is specified by any one set of values of the outputs. For a query like `unitPriceByDay`, the `ValueCursor` of its `Cursor` has only one value, and therefore only one position, at a time for any one position of the root `CompoundCursor`.

The following table illustrates one possible display of the data from `queryCursor`. It is a crosstab view with four columns and five rows. In the left column are the day values. In the top row are the product values. In each of the intersecting cells of the crosstab is the price of the product on the day.

	Product		
Day	**815**	**1050**	**2055**
01-JAN-00	58	24	24
01-APR-00	59	24	25
01-JUL-00	59	25	25
01-OCT-00	61	25	26

A `CompoundCursor` coordinates the positions of its `ValueCursor` objects relative to each other. The current position of the `CompoundCursor` specifies the current positions of its descendent `ValueCursor` objects. Example 8–1 sets the position of `queryCursor` and then gets the current values and the positions of the child `Cursor` objects.

Example 8–1 Setting the CompoundCursor Position and Getting the Current Values

```
CompoundCursor rootCursor = (CompoundCursor) queryCursor;
ValueCursor baseValueCursor = rootCursor.getValueCursor();
List outputs = rootCursor.getOutputs();
ValueCursor output1 = (ValueCursor) outputs.get(0);
ValueCursor output2 = (ValueCursor) outputs.get(1);
int pos = 5;
root.setPosition(pos);
System.out.println("CompoundCursor position set to " + pos + ".");
System.out.println("CC position = " + rootCursor.getPosition() + ".");
System.out.println("Output 1 position = " + output1.getPosition() +
                   ", value = " + output1.getCurrentValue());
System.out.println("Output 2 position = " + output2.getPosition() +
                   ", value = " + output2.getCurrentValue());
System.out.println("VC position = " + baseValueCursor.getPosition() +
                   ", value = " + baseValueCursor.getCurrentValue());
```

Example 8–1 displays the following:

```
CompoundCursor position set to 5.
CC position = 5.
Output 1 position = 2, value = 01-APR-00
Output 2 position = 2, value = 1050
VC position = 1, value = 24
```

The positions of `queryCursor` are symmetric in that the result set for `unitPriceByDay` always has three product values for each time value. The `ValueCursor` for `productSel`, therefore, always has three positions for each value of the `timeSel` `ValueCursor`. The `timeSel` output `ValueCursor` is slower varying than the `productSel` `ValueCursor`.

In an asymmetric case, however, the number of positions in a `ValueCursor` is not always the same relative to its slower varying output. For example, if the price of units for product 2055 on October 1, 2000 were null because that product was no longer being purchased by that date, and if null values were suppressed in the query, then `queryCursor` would only have eleven positions. The `ValueCursor` for `productSel` would only have two positions when the position of the `ValueCursor` for `timeSel` was 4.

Example 8–2 produces an asymmetric result set by using a revision of the query from "Structure of a Cursor" on page 8-5. The result set of the revised query specifies products by price on a day. The base values of `productByPriceOnDay` are the values from `productSel` as specified by the values of `unitPrice` and `timeSel`.

Because `productByPriceOnDay` is a derived `Source`, this example prepares and commits the current `Transaction`. The `TransactionProvider` in the example is `tp`. For information on `Transaction` objects, see Chapter 7.

The example creates a `Cursor` for `productByPriceOnDay`, loops through the positions of the `CompoundCursor`, gets the position and current value of each child `ValueCursor` object, and displays the positions and values.

Example 8–2 Positions in an Asymmetric Query

```
// Create the query
productByPriceOnDay = productSel.join(unitPrice).join(timeSel);

//Prepare and commit the current Transaction.
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create the Cursor. The DataProvider is dp.
CursorManagerSpecification cursorMngrSpec =
                  dp.createCursorManagerSpecification(productByPriceOnDay);
CursorManager cursorManager = dp.createCursorManager(cursorMngrSpec);
Cursor queryCursor2 = cursorManager.createCursor();

// Get the ValueCursor and the outputs
CompoundCursor rootCursor = (CompoundCursor) queryCursor2;
ValueCursor baseValueCursor = rootCursor.getValueCursor();
List outputs = rootCursor.getOutputs();
ValueCursor output1 = (ValueCursor) outputs.get(0);
ValueCursor output2 = (ValueCursor) outputs.get(1);

// Get the positions and values and display them.
System.out.println("   CC \t\tOutput 1 \tOutput 2 \tVC");
System.out.println("position \tposition:value " +
                   "\tposition:value \tposition:value");
do {
System.out.println("      " + root.getPosition() +
                   "\t\t   " + output1.getPosition() +
                   "  :  " + output1.getCurrentValue() +
                   "\t   " + output2.getPosition() +
                   "  :  "  + output2.getCurrentValue() +
                   "\t   " + baseValueCursor.getPosition() +
                   "  :  " + baseValueCursor.getCurrentValue());
}
while(queryCursor2.next();
```

Example 8–2 displays the following:

```
   CC        Output 1           Output 2           VC
position     position:value     position:value     position:value
    1          1  :  01-JAN-00       1  :  58            1  :  815
    2          1  :  01-JAN-00       2  :  24            1  :  1050
    3          1  :  01-JAN-00       2  :  24            2  :  2055
    4          2  :  01-APR-00       1  :  59            1  :  815
    5          2  :  01-APR-00       2  :  24            1  :  1050
    6          2  :  01-APR-00       3  :  25            1  :  2055
    7          3  :  01-JUL-00       1  :  59            1  :  815
    8          3  :  01-JUL-00       2  :  25            1  :  1050
    9          3  :  01-JUL-00       2  :  25            2  :  2055
    10         4  :  01-OCT-00       1  :  61            1  :  815
    11         4  :  01-OCT-00       2  :  25            1  :  1050
    12         4  :  01-OCT-00       3  :  26            1  :  2055
```

The ValueCursor with unitPrice values (output 2) has only two positions for 01-JAN-00 and 01-JUL-00 because it has only two different values for those days. The prices of two of the products are the same on those two days: 24 for products 1050 and 2055 on January 1, 2000 and 25 for those same two products on July 1, 2000. The base ValueCursor for queryCursor2 has two positions when the timeSel value is 01-JAN-00 or 01-JUL-00 because each of the unitPrice values for those days is not unique.

About the Parent Starting and Ending Positions in a Cursor

To effectively manage the display of the data that you get from a CompoundCursor, you sometimes need to know how many faster varying values exist for the current slower varying value. For example, suppose that you are displaying in a crosstab one row of values from an edge of a cube, then you might want to know how many columns to draw in the display for the row.

To determine how many faster varying values exist for the current value of a child Cursor, you find the starting and ending positions of that current value in the parent Cursor. Subtract the starting position from the ending position and then add 1, as in the following.

```
long span = (cursor.getParentEnd() - cursor.getParentStart()) + 1;
```

The result is the *span* of the current value of the child Cursor in its parent Cursor, which tells you how many values of the fastest varying child Cursor exist for the current value. Calculating the starting and ending positions is costly in time and

computing resources, so you should only specify that you want those calculations performed when your application needs the information.

An Oracle OLAP API `Cursor` enables your application to have only the data that it is currently displaying actually present on the client computer. For information on specifying the amount of data for a `Cursor`, see "About Fetch Sizes and Fetch Blocks" on page 8-27.

From the data on the client computer, however, you cannot determine at what position of its parent `Cursor` the current value of a child `Cursor` begins or ends. To get that information, you use the `getParentStart` and `getParentEnd` methods of a `Cursor`.

For example, suppose your application has a `Source` named `cube` that represents a cube that has an asymmetric edge. The cube has four outputs. The `cube Source` defines products with sales amounts greater than $5,000 purchased by customers in certain cities during the first three months of the calendar year 2000. The products were sold through the direct sales channel (S) during a television promotion (TV).

You create a `Cursor` for that `Source` and call it `cubeCursor`. The `CompoundCursor cubeCursor` has the following child `Cursor` objects:

- output 1, a `ValueCursor` for the promotion values
- output 2, a `ValueCursor` for the channel values
- output 3, a `ValueCursor` for the time values
- output 4, a `ValueCursor` for the customer values
- The base `ValueCursor`, which has values that are the products with sales amounts over $5,000.

Figure 8–5 illustrates the parent, `cubeCursor`, with the values of its child `Cursor` objects layered horizontally. The slowest varying output, with the promotion values, is at the top and the fastest varying child, with the product values, is at the bottom. The only portion of the edge that you are currently displaying in the user interface is the block between positions 7 and 9 of `cubeCursor`, which is shown within the bold border. The positions, 1 through 10, of `cubeCursor` appear above the top row.

Figure 8–5 Values of ValueCursor Children of cubeCursor

1	2	3	4	5	6	7	8	9	10
TV									
S									
2000-01			2000-02				2000-03		
Bonn		London	Bonn		London	Paris	Bonn	London	
1050	2055	815	1050	1555	935	1050	935	1050	3690

The current value of the output `ValueCursor` for the time `Source` is 2000-02. You cannot determine from the data within the block that the starting and ending positions of the current value, 2000-02, in the parent, `cubeCursor`, are 4 and 7, respectively.

The `cubeCursor` from the previous figure is shown again in Figure 8–6, this time with the range of the positions of the parent, `cubeCursor`, for each of the values of the child `Cursor` objects. By subtracting the smaller value from the larger value and adding one, you can compute the span of each value. For example, the span of the time value 2000-02 is (7 - 4 + 1) = 4.

Figure 8–6 The Range of Positions of the Child Cursor Objects of cubeCursor

<table>
<tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td></tr>
<tr><td colspan="10">1-10</td></tr>
<tr><td colspan="10">1-10</td></tr>
<tr><td colspan="3">1 to 3</td><td colspan="4">4 to 7</td><td colspan="3">8 to 10</td></tr>
<tr><td colspan="2">1 to 2</td><td>3 to 3</td><td colspan="2">4 to 5</td><td>6 to 6</td><td>7 to 7</td><td>8 to 8</td><td colspan="2">9 to 10</td></tr>
<tr><td>1 to 1</td><td>2 to 2</td><td>3 to 3</td><td>4 to 4</td><td>5 to 5</td><td>6 to 6</td><td>7 to 7</td><td>8 to 8</td><td>9 to 9</td><td>10 to 10</td></tr>
</table>

To specify that you want Oracle OLAP to calculate the starting and ending positions of a value of a child `Cursor` in its parent `Cursor`, call the `setParentStartCalculationSpecified` and `setParentEndCalculationSpecified` methods on the `CursorSpecification` corresponding to the `Cursor`. You can determine whether calculating the starting or ending positions is specified by calling the `isParentStartCalculationSpecified` or `isParentEndCalculationSpecified` methods on the `CursorSpecification`. For an example of specifying these calculations, see Chapter 9.

What is the Extent of a Cursor?

The extent of a `Cursor` is the total number of elements it contains relative to any slower varying outputs. Figure 8–7 illustrates the number of positions of each child `Cursor` of `cubeCursor` relative to the value of its slower varying output. The child `Cursor` objects are layered horizontally with the slowest varying output at the top.

The total number of elements in `cubeCursor` is 10 so the extent of `cubeCursor` is therefore 10. That number is above the top row of the figure. The top row is the `ValueCursor` for the promotion value and the next row down is the `ValueCursor` for the channel value. The extent of each of those `ValueCursor` objects is 1 because they each have only one value.

The third row down represents the time values. Its extent is 3, since there are 3 months values. The next row down is the `ValueCursor` for the customers by city. The extent of its elements depends on the value of the slower varying output, which is time. The extent of the customers `ValueCursor` for the first month is 2, for the second month it is 3, and for the third month it is 2.

The bottom row is the base `ValueCursor` for the `cubeCursor` `CompoundCursor`. Its values are products. The extent of the elements of the products `ValueCursor` depends on the values of the customers `ValueCursor` and the time `ValueCursor`. For example, since two products values are specified by the first set of month and city values (1050 and 2055 for Bonn in 2000-01), the extent of the products `ValueCursor` for that set is 2. For the second set of values for customers and times (2000-10, London), the extent of the products `ValueCursor` is 1, and so on.

Figure 8–7 The Number of Elements of the Child Cursor Objects of cubeCursor

10

<table>
<tr><td colspan="10">1</td></tr>
<tr><td colspan="10">1</td></tr>
<tr><td colspan="3">1</td><td colspan="4">2</td><td colspan="3">3</td></tr>
<tr><td colspan="2">1</td><td>2</td><td colspan="2">1</td><td>2</td><td>3</td><td>1</td><td colspan="2">2</td></tr>
<tr><td>1</td><td>2</td><td>1</td><td>1</td><td>2</td><td>1</td><td>1</td><td>1</td><td>1</td><td>2</td></tr>
</table>

The extent is information that you can use, for example, to display the correct number of columns or correctly-sized scroll bars. The extent, however, can be expensive to calculate. For example, a `Source` that represents a cube might have four outputs. Each output might have hundreds of values. If all null values and zero values of the measure for the sets of outputs are eliminated from the result set, then to calculate the extent of the `CompoundCursor` for the `Source`, Oracle OLAP must traverse the entire result space before it creates the `CompoundCursor`. If you do not specify that you wants the extent calculated, then Oracle OLAP only needs to traverse the sets of elements defined by the outputs of the cube as specified by the fetch size of the `Cursor` and as needed by your application.

To specify that you want Oracle OLAP to calculate the extent for a `Cursor`, call the `setExtentCalculationSpecified` method on the `CursorSpecification` corresponding to the `Cursor`. You can determine whether calculating the extent is specified by calling the `isExtentCalculationSpecified` method on the `CursorSpecification`. For an example of specifying the calculation of the extent of a `Cursor`, see Chapter 9.

About Fetch Sizes and Fetch Blocks

An OLAP API `Cursor` represents the entire result set for a `Source`. The `Cursor` is a ***virtual*** `Cursor`, however, because it retrieves only a portion of the result set at a time from Oracle OLAP. A `CursorManager` manages a virtual `Cursor` and retrieves results from Oracle OLAP as your application needs it. By managing the virtual `Cursor`, the `CursorManager` relieves your application of a substantial burden.

The amount of data that a `Cursor` retrieves in a single fetch operation is determined by the ***fetch size*** specified for the `Cursor`. For a `CompoundCursor`, the amount of data fetched in a single operation is the product of the fetch sizes of all of its descendent `ValueCursor` objects. The total set of values retrieved in a single fetch is the ***fetch block*** for the `Cursor`. You specify fetch sizes in order to limit the amount of data your application needs to cache on the local computer and to maximize the efficiency of the fetch by customizing it to meet the needs of your method of displaying of the data.

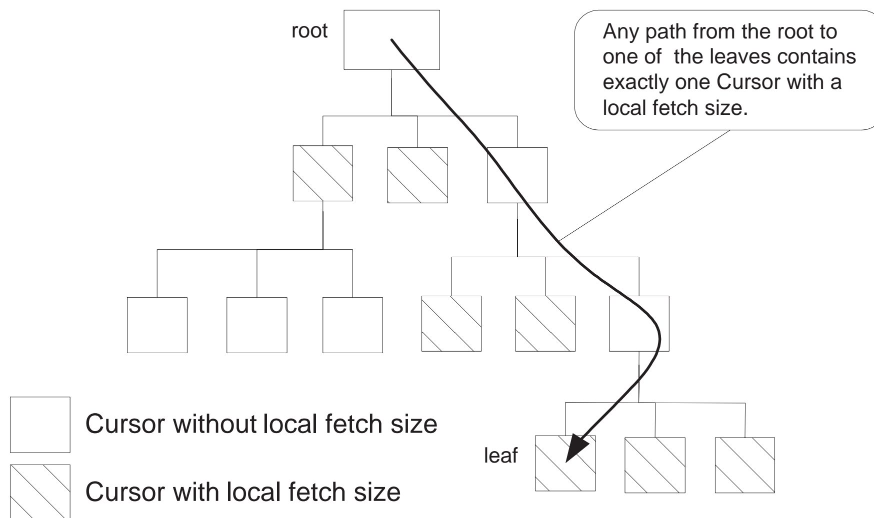
When you create a `CursorManagerSpecification` for a `Source`, as the first step in creating a `Cursor`, Oracle OLAP specifies a default fetch size on the root `CursorSpecification` of the `CursorManagerSpecification`. By calling methods on the `CursorSpecification` objects of the `CursorManagerSpecification`, you can specify a default fetch size or specify setting the fetch size at other levels of a `CompoundCursor`.

If the fetch size is specified on a `CursorSpecification`, then you can get or set the fetch size for the corresponding `Cursor` by calling the `getFetchSize` or `setFetchSize` method on that `Cursor`. For a `CompoundCursor`, you can set different fetch sizes for child `Cursor` objects at different levels in the outputs.

A `Cursor` has a ***local fetch size*** if the size of the fetch block is specified for that `Cursor`. Not all of the `Cursor` objects in a `CompoundCursor` can have local fetch sizes. The structure of a `CompoundCursor` is like a tree, with the hierarchy of `Cursor` objects starting at the topmost (root) `Cursor` and going down through all the child `Cursor` objects. Any path through the hierarchy, starting from the root and going down to a leaf `ValueCursor`, can contain one, and only one, `Cursor` with a local fetch size. Specifying the fetch size on a parent `Cursor` affects all of the child `Cursor` objects of that parent. This means that a fetch block can contain no more than the number of elements of each child `Cursor` specified by the fetch size.

Figure 8–8 shows an example of a path through the hierarchy of a `Cursor` tree in which the `Cursor` objects with local fetch sizes are shaded.

Figure 8–8 A Local Fetch Size Path Through a Cursor Hierarchy



About Determining the Shape of a Fetch Block

In a CompoundCursor, the levels at which you set the fetch sizes determine the *shape* of the fetch block of the CompoundCursor. The optimal fetch block for a CompoundCursor depends on the way you intend to navigate the Cursor and display the data. After determining how to display the data, you should do the following:

- Specify a fetch block that is large enough to contain all the data required for the portion of the result set that you are displaying in the user interface. For example, if you display the data in a table and the size of the window means that 25 rows are visible at a time, then the fetch block should contain at least 25 rows. If it is any smaller than this, the Cursor needs to make multiple trips to Oracle OLAP to fill the display.
- Specify fetch sizes on the Cursor objects that you use to loop through the result set. For example, for a table view, set fetch sizes on the root Cursor and for a crosstab view, set fetch sizes on the child Cursor objects.
- Keep the product of all of the fetch sizes relatively small because the product determines the total number of cells in the fetch block. If the product of all the fetch sizes is too large, then you lose the advantages of the virtual Cursor.

For examples of specifying fetch sizes and fetch blocks for different displays, see Chapter 9.

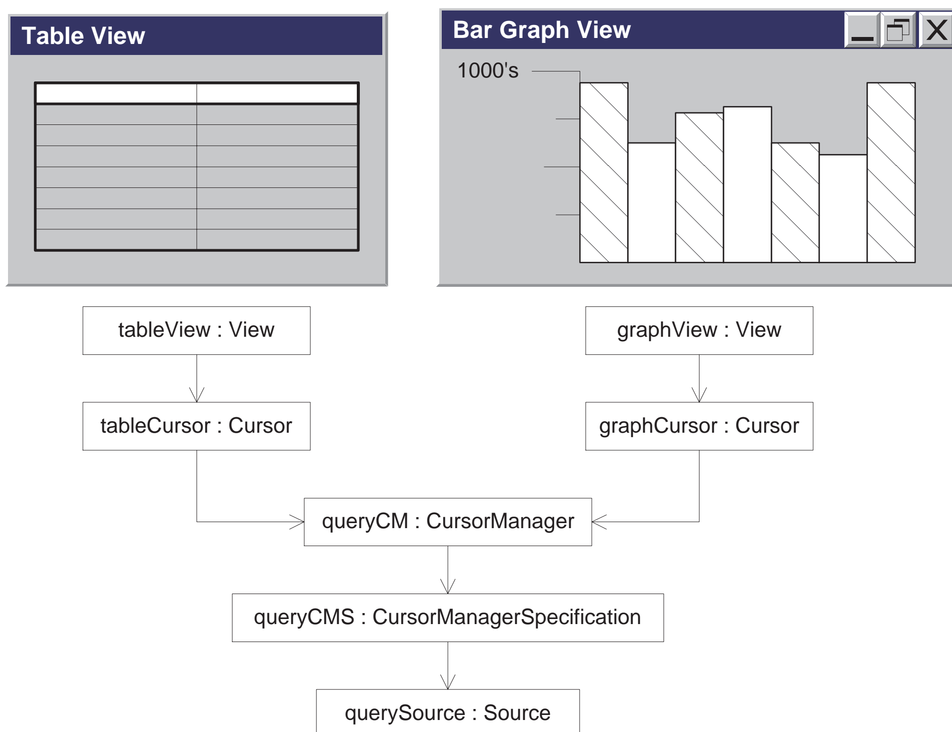
About Sharing Fetch Blocks

You can create two or more Cursor objects from the same CursorManager and use both Cursor objects simultaneously. The Cursor objects can share the data managed by the CursorManager, rather than having separate data caches, because the shape of the fetch blocks is the same for both Cursor objects. The shape of the fetch blocks is determined by the levels of the CursorManagerSpecification on which the fetch size is specified.

An example is an application that displays the results of a query to the user as both a table and a graph. The application creates a CursorManagerSpecification for a Source and then creates a CursorManager for the CursorManagerSpecification. The application creates two separate Cursor objects from the same CursorManager, one for a table view and one for a graph view. The two views share the same query and display the same data, just in

different formats. Figure 8–9 illustrates the relationship between the `Source`, the `Cursor` objects, and the views.

Figure 8–9 A Source and Two Cursors for Different Views of Its Values



9

Retrieving Query Results

This chapter describes how to retrieve the results of a query with an Oracle OLAP API `Cursor` and how to gain access to those results. This chapter also describes how to customize the behavior of a `Cursor` to fit your method of displaying the results. For information on the class hierarchies of `Cursor` and its related classes, and for information on the `Cursor` concepts of position, fetch size, and extent, see Chapter 8.

This chapter includes the following topics:

- Retrieving the Results of a Query
- Navigating a CompoundCursor for Different Displays of Data
- Specifying the Behavior of a Cursor
- Calculating Extent and Starting and Ending Positions of a Value
- Specifying Fetch Sizes and Fetch Blocks

Retrieving the Results of a Query

A query is an OLAP API Source that specifies the data that you want to retrieve from Oracle OLAP and any calculations you want Oracle OLAP to perform on that data. A Cursor is the object that retrieves, or *fetches*, the result set specified by a Source. Creating a Cursor for a Source involves the following steps:

1. Get a primary Source from an MdmObject or create a derived Source through operations on a DataProvider or a Source. For information on getting or creating Source objects, see Chapter 5.
2. If the Source is a derived Source, prepare and commit the Transaction in which you created the Source. To prepare and commit the Transaction, call the prepareCurrentTransaction and commitCurrentTransaction methods on your TransactionProvider. For more information on preparing and committing a Transaction, see Chapter 7. If the Source is a primary Source, then you do not need to prepare and commit the Transaction.
3. Create a CursorManagerSpecification by calling the createCursorManagerSpecification method on your DataProvider and passing that method the Source.
4. Create a SpecifiedCursorManager by calling the createCursorManager method on your DataProvider and passing that method the CursorManagerSpecification. If the Source for the CursorManagerSpecification has one or more inputs, then you must also pass an array of Source objects that provides a Source for each input.
5. Create a Cursor by calling the createCursor method on the CursorManager. If you created the CursorManager with an array of input Source objects, then you must also pass an array of CursorInput objects that provides a value for each input Source.

Example 9–1 creates a Cursor for the derived Source named querySource. The example uses a TransactionProvider named tp and a DataProvider named dp. The example creates a CursorManagerSpecification named cursorMngrSpec, a SpecifiedCursorManager named cursorMngr, and a Cursor named queryCursor.

Finally, the example closes the SpecifiedCursorManager. When you have finished using the Cursor, you should close the SpecifiedCursorManager to

free resources.

Example 9–1 Creating a Cursor

```
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  System.out.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();
CursorManagerSpecification cursorMngrSpec =
                 dp.createCursorManagerSpecification(querySource);
SpecifiedCursorManager cursorMngr =
                           dp.createCursorManager(cursorMngrSpec);
Cursor queryCursor = cursorMngr.createCursor();

// ... Use the Cursor in some way, such as to display its values.

cursorMngr.close();
```

Getting Values from a Cursor

The `Cursor` interface encapsulates the notion of a *current position* and has methods for moving the current position. The `ValueCursor` and `CompoundCursor` interfaces extend the `Cursor` interface. The Oracle OLAP API has implementations of the `ValueCursor` and `CompoundCursor` interfaces. Calling the `createCursor` method on a `CursorManager` returns either a `ValueCursor` or a `CompoundCursor` implementation, depending on the `Source` for which you are creating the `Cursor`.

A `ValueCursor` is returned for a `Source` that has a single set of values. A `ValueCursor` has a value at its current position, and it has methods for getting the value at the current position.

A `CompoundCursor` is created for a `Source` that has more than one set of values, which is a `Source` that has one or more outputs. Each set of values of the `Source` is represented by a child `ValueCursor` of the `CompoundCursor`. A `CompoundCursor` has methods for getting its child `Cursor` objects.

The structure of the `Source` determines the structure of the `Cursor`. A `Source` can have nested outputs, which occurs when one or more of the outputs of the `Source` is itself a `Source` with outputs. If a `Source` has a nested output, then the

CompoundCursor for that Source has a child CompoundCursor for that nested output.

The CompoundCursor coordinates the positions of its child Cursor objects. The current position of the CompoundCursor specifies one set of positions of its child Cursor objects.

For an example of a Source that has only one level of output values, see Example 9–4. For an example of a Source that has nested output values, see Example 9–5.

An example of a Source that represents a single set of values is one returned by the getSource method on an MdmDimension, such as an MdmDimension that represents a hierarchical list of product values. Creating a Cursor for that Source returns a ValueCursor. Calling the getCurrentValue method returns the product value at the current position of that ValueCursor.

Example 9–2 gets the Source from mdmProductHier, which is an MdmDimension that represents product values, and creates a Cursor for that Source. The example sets the current position to the fifth element of the ValueCursor and gets the product value from the Cursor. The example then closes the CursorManager. In the example, dp is the DataProvider.

Example 9–2 Getting a Single Value from a ValueCursor

```
Source productSource = mdmProductHier.getSource();
// Because productSource is a primary Source, you do not need to
// prepare and commit the current Transaction.
CursorManagerSpecification cursorMngrSpec =
               dp.createCursorManagerSpecification(productSource);
SpecifiedCursorManager cursorMngr =
                           dp.createCursorManager(cursorMngrSpec);
Cursor productCursor = cursorMngr.createCursor();
// Cast the Cursor to a ValueCursor.
ValueCursor productValues = (ValueCursor) productCursor;
// Set the position to the fifth element of the ValueCursor.
productValues.setPosition(5);
```

```
// Product values are strings. Get the String value at the current
// position.
String value = productValues.getCurrentString();

// Do something with the value, such as display it...

// Close the SpecifiedCursorManager.
cursorMngr.close();
```

Example 9–3 uses the same `Cursor` as Example 9–2. Example 9–3 uses a `do...while` loop and the `next` method of the `ValueCursor` to move through the positions of the `ValueCursor`. The `next` method begins at a valid position and returns `true` when an additional position exists in the `Cursor`. It also advances the current position to that next position.

The example sets the position to the first position of the `ValueCursor`. The example loops through the positions and uses the `getCurrentValue` method to get the value at the current position.

Example 9–3 Getting All of the Values from a ValueCursor

```
// productValues is the ValueCursor for productSource
productValues.setPosition(1);
do {
   System.out.println(productValues.getCurrentValue);
}
while(productValues.next());
```

The values of the result set represented by a `CompoundCursor` are in the child `ValueCursor` objects of the `CompoundCursor`. To get those values, you must get the child `ValueCursor` objects from the `CompoundCursor`.

An example of a `CompoundCursor` is one that is returned by calling the `createCursor` method on a `CursorManager` for a `Source` that represents the values of a measure as specified by selected values from the dimensions of the measure.

Example 9–4 uses a `Source`, named `salesAmount`, that results from calling the `getSource` method on an `MdmMeasure` that represents monetary amounts for sales. The dimensions of the measure are `MdmDimension` objects representing products, customers, times, channels, and promotions. This example uses `Source` objects that represent selected values from those dimensions. The names of those `Source` objects are `prodSel`, `custSel`, `timeSel`, `chanSel`, and `promoSel`. The

creation of the Source objects representing the measure and the dimension selections is not shown.

Example 9–4 joins the dimension selections to the measure, which results in a Source named salesForSelections. It creates a Cursor, named salesForSelCursor, for salesForSelections, casts the Cursor to a CompoundCursor, named salesCompndCrsr, and gets the base ValueCursor and the outputs from the CompoundCursor. Each output is a ValueCursor, in this case. The outputs are returned in a List. The order of the outputs in the List is the inverse of the order in which the dimensions were joined to the measure. In the example, dp is the DataProvider and tp is the TransactionProvider.

Example 9–4 Getting ValueCursor Objects from a CompoundCursor

```
Source salesForSelections = salesAmount.join(prodSel)
                                       .join(custSel)
                                       .join(timeSel)
                                       .join(chanSel)
                                       .join(promoSel);
// Prepare and commit the current Transaction
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a Cursor for salesForSelections
CursorManagerSpecification cursorMngrSpec =
            dp.createCursorManagerSpecification(salesForSelections);
SpecifiedCursorManager cursorMngr =
                             dp.createCursorManager(cursorMngrSpec);
Cursor salesForSelCursor = cursorMngr.createCursor();

// Cast salesForSelCursor to a CompoundCursor
CompoundCursor salesCompndCrsr = (CompoundCursor) salesValues;

// Get the base ValueCursor
ValueCursor specifiedSalesVals = salesCompndCrsr.getValueCursor();
```

```
// Get the outputs
List outputs = salesCompndCrsr.getOutputs();
ValueCursor promoSelVals = (ValueCursor) outputs.get(0);
ValueCursor chanSelVals = (ValueCursor) outputs.get(1);
ValueCursor timeSelVals = (ValueCursor) outputs.get(2);
ValueCursor custSelVals = (ValueCursor) outputs.get(3);
ValueCursor prodSelVals = (ValueCursor) outputs.get(4);

// You can now get the values from the ValueCursor objects.
// When you have finished using the Cursor objects, close the
// SpecifiedCursorManager.
cursorMngr.close()
```

Example 9–5 uses the same sales amount measure as Example 9–4, but it joins the dimension selections to the measure differently. Example 9–5 joins two of the dimension selections together. It then joins the result to the `Source` that results from joining the single dimension selections to the measure. The resulting `Source`, `salesForSelections`, represents a query has nested outputs, which means it has more than one level of outputs.

The `CompoundCursor` that this example creates for `salesForSelections` therefore also has nested outputs. The `CompoundCursor` has a child base `ValueCursor` and as its outputs has three child `ValueCursor` objects and one child `CompoundCursor`.

Example 9–5 joins the selection of promotion dimension values, `promoSel`, to the selection of channel dimension values, `chanSel`. The result is `chanByPromoSel`, a `Source` that has channel values as its base values and promotion values as the values of its output. The example joins to `salesAmount` the selections of product, customer, and time values, and then joins `chanByPromoSel`. The resulting query is represented by `salesForSelections`.

The example prepares and commits the current `Transaction` and creates a `Cursor`, named `salesForSelCursor`, for `salesForSelections`.

The example casts the `Cursor` to a `CompoundCursor`, named `salesCompndCrsr`, and gets the base `ValueCursor` and the outputs from it. In the example, `dp` is the `DataProvider` and `tp` is the `TransactionProvider`.

Example 9–5 Getting Values from a CompoundCursor with Nested Outputs

```
// ...in someMethod...
Source chanByPromoSel = chanSel.join(promoSel);
Source salesForSelections = salesAmount.join(prodSel)
                                       .join(custSel)
                                       .join(timeSel)
                                       .join(chanByPromoSel);

// Prepare and commit the current Transaction
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a Cursor for salesForSelections
CursorManagerSpecification cursorMngrSpec =
            dp.createCursorManagerSpecification(salesForSelections);
SpecifiedCursorManager cursorMngr =
                             dp.createCursorManager(cursorMngrSpec);
Cursor salesForSelCursor = cursorMngr.createCursor();

// Send the Cursor to a method that does different operations
// depending on whether the Cursor is a CompoundCursor or a
// ValueCursor.
printCursor(salesForSelCursor);
cursorMngr.close();
// ...the remaining code of someMethod...
```

```
// The printCursor method has a do...while loop that moves through the positions
// of the Cursor passed to it. At each position, the method prints the number of
//  the iteration through the loop and then a colon and a space. The output
// object is a PrintWriter. The method calls the private _printTuple method and
// then prints a new line. A "tuple" is the set of output ValueCursor values
// specified by one position of the parent CompoundCursor. The method prints one
// line for each position of the parent CompoundCursor.
public void printCursor(Cursor rootCursor) {
  int i = 1;
  do {
     output.print(i++ + ": ");
     _printTuple(rootCursor);
     output.print("\n");
     output.flush();
  }
  while(rootCursor.next());
}

// If the Cursor passed to the _printTuple method is a ValueCursor,
// the method prints the value at the current position of the ValueCursor.
// If the Cursor passed in is a CompoundCursor, the method gets the
// outputs of the CompoundCursor and iterates through the outputs,
// recursively calling itself for each output. The method then gets the
// base ValueCursor of the CompoundCursor and calls itself again.
private void _printTuple(Cursor cursor) {
  if(cursor instanceof CompoundCursor) {
    CompoundCursor compoundCursor = (CompoundCursor)cursor;
    // Put an open parenthesis before the value of each output
    output.print("(");
    Iterator iterOutputs = compoundCursor.getOutputs().iterator();
    Cursor output = (Cursor)iterOutputs.next();
    _printTuple(output);
    while(iterOutputs.hasNext()) {
      // Put a comma after the value of each output
      output.print(",");
      _printTuple((Cursor)iterOutputs.next());
    }
     // Put a comma after the value of the last output
    output.print(",");
    // Get the base ValueCursor
    _printTuple(compoundCursor.getValueCursor());
```

```
    // Put a close parenthesis after the base value to indicate
    // the end of the tuple.
        output.print(")");
   }
   else if(cursor instanceof ValueCursor) {
     ValueCursor valueCursor = (ValueCursor) cursor;
     if (valueCursor.hasCurrentValue())
       print(valueCursor.getCurrentValue());
     else                       // If this position has a null value
       print("NA");
   }
 }
```

Navigating a CompoundCursor for Different Displays of Data

With methods on a `CompoundCursor` you can easily move through, or navigate, its structure and get the values from its `ValueCursor` descendents. Data from a multidimensional OLAP query is often displayed in a crosstab format, or as a table or a graph.

To display the data for multiple rows and columns, you loop through the positions at different levels of the `CompoundCursor` depending on the needs of your display. For some displays, such as a table, you loop through the positions of the parent `CompoundCursor`. For other displays, such as a crosstab, you loop through the positions of the child `Cursor` objects.

To display the results of a query in a table view, in which each row contains a value from each output `ValueCursor` and from the base `ValueCursor`, you determine the position of the top-level, or root, `CompoundCursor` and then iterate through its positions. Example 9–6 displays only a portion of the result set at one time. It creates a `Cursor` for a `Source` that represents a query that is based on a measure that has unit cost values. The dimensions of the measure are the product and time dimensions. The creation of the primary `Source` objects and the derived selections of the dimensions is not shown.

The example joins the `Source` objects representing the dimension value selections to the `Source` representing the measure. It prepares and commits the current `Transaction` and then creates a `Cursor`. It casts the `Cursor` to a `CompoundCursor`. The example sets the position of the `CompoundCursor`, iterates through twelve positions of the `CompoundCursor`, and prints out the values specified at those positions. The `TransactionProvider` is `tp` and the `DataProvider` is `dp`. The `output` object is a `PrintWriter`.

Example 9–6 Navigating for a Table View

```
Source unitPriceByDay = unitPrice.join(productSel)
                                 .join(timeSel);
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a Cursor for unitPriceByDay
CursorManagerSpecification cursorMngrSpec =
            dp.createCursorManagerSpecification(unitPriceByDay);
SpecifiedCursorManager cursorMngr =
                             dp.createCursorManager(cursorMngrSpec);
Cursor unitPriceByDayCursor = cursorMngr.createCursor();

// Cast the Cursor to a CompoundCursor
CompoundCursor rootCursor = (CompoundCursor) unitPriceByDayCursor;

// Determine a starting position and the number of rows to display
int start = 7;
int numRows = 12;

// Iterate through the specified positions of the root CompoundCursor.
// Assume that the Cursor contains at least (start + numRows) positions.
for(int pos = start; pos < start + numRows; pos++) {
   // Set the position of the root CompoundCursor
   rootCursor.setPosition(pos);
   // Print the values of the output ValueCursors
   output.print(rootCursor.getOutputs().get(0).getCurrentValue() + "\t");
   output.print(rootCursor.getOutputs().get(1).getCurrentValue() + "\t");
   // Print the value of the base ValueCursor and a new line
   output.print(rootCursor.getValueCursor().getCurrentValue() + "\n");
   output.flush();
};
cursorMngr.close();
```

If the time selection for the query has eight values, such as the first day of each calendar quarter for the years 1999 and 2000, and the product selection has three values, then the result set of the `unitPriceByDay` query has twenty-four

positions. Example 9–6 displays something like the following table, which has the values specified by positions 7 through 18 of the `CompoundCursor`.

```
01-JUL-99    815     57
01-JUL-99    1050    23
01-JUL-99    2055    22
01-OCT-99    815     56
01-OCT-99    1050    24
01-OCT-99    2055    21
01-JAN-00    815     58
01-JAN-00    1050    24
01-JAN-00    2055    24
01-APR-00    815     59
01-APR-00    1050    24
01-APR-00    2055    25
```

Example 9–7 uses the same query as Example 9–6. In a crosstab view, the first row is column headings, which are the values from `timeSel` in this example. The output for `timeSel` is the faster varying output because the `timeSel` dimension selection was joined to the measure first. The remaining rows begin with a row heading. The row headings are values from the slower varying output, which is `productSel`. The remaining positions of the rows, under the column headings, contain the `unitPrice` values specified by the set of the dimension values.

To display the results of a query in a crosstab view, you specify the positions of the children of the top-level `CompoundCursor` and then iterate through their positions. Example 9–7 gets the values but does not include code for putting the values in the appropriate cells of the crosstab display.

Example 9–7 Navigating for a Crosstab View without Pages

```
Source unitPriceByDay = unitPrice.join(productSel)
                                 .join(timeSel);
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();
```

```
// Create a Cursor for unitPriceByDay
CursorManagerSpecification cursorMngrSpec =
            dp.createCursorManagerSpecification(unitPriceByDay);
SpecifiedCursorManager cursorMngr =
                         dp.createCursorManager(cursorMngrSpec);
Cursor unitPriceByDayCursor = cursorMngr.createCursor();

// Cast the Cursor to a CompoundCursor
CompoundCursor rootCursor = (CompoundCursor) unitPriceByDayCursor;

// Determine a starting position and the number of rows to display.
// colStart is the position in columnCursor at which the current
// display starts and rowStart is the position in rowCursor at
// which the current display starts.
int colStart = 1;
int rowStart = 1;
String productValue;
String timeValue;
double price;
int numProducts = 3;
int numDays = 12;

// Get the outputs and the ValueCursor
CompoundCursor rootCursor = (CompoundCursor) unitPriceByDayCursor;
List outputs = rootCursor.getOutputs();
// The first output has the values of timeSel, the slower varying output
ValueCursor rowCursor = (ValueCursor) outputs.get(0);
// The second output has the faster varying values of productSel
ValueCursor columnCursor = (ValueCursor) outputs.get(1);
ValueCursor unitPriceValues = rootCursor.getValueCursor();// Prices

// Loop through positions of the faster varying output Cursor
for(int pPos = colStart; pPos < colStart + numProducts; pPos++) {
  columnCursor.setPosition(pPos);
  // Loop through positions of the slower varying output Cursor
  for(int tPos = rowStart; tPos < rowStart + numDays; tPos++) {
    rowCursor.setPosition(tPos);
    // Get the values. Sending the values to the appropriate
    // display mechanism is not shown.
    productValue = columnCursor.getCurrentString();
    timeValue = rowCursor.getCurrentString();
    price = unitPriceValues.getCurrentDouble();
  }
}
cursorMngr.close();
```

Figure 9–1 is crosstab view of the values from the result set specified by the `unitPriceByDay` query.

Figure 9–1 Crosstab View of the Result Set Specified by unitPriceByDay

	815	1050	2055
01-JAN-99	56	22	21
01-APR-99	57	22	21
01-JUL-99	57	23	22
01-OCT-99	56	24	21
01-JAN-00	58	24	24
01-APR-00	59	24	25
01-JUL-00	59	25	25
01-OCT-00	61	25	26

Example 9–8 creates a `Source` that is based on a sales amount measure. The dimensions of the measure are the customer, product, time, channel, and promotion dimensions. The `Source` objects for the dimensions represent selections of the dimension values. The creation of those `Source` objects is not shown.

The query that results from joining the dimension selections to the measure `Source` represents total sales amount values as specified by the values of its outputs.

The example creates a `Cursor` for the query and then sends the `Cursor` to the `printAsCrosstab` method, which prints the values from the `Cursor` in a crosstab. That method calls other methods that print page, column, and row values.

The fastest varying output of the `Cursor` is the selection of customers, which has three values that specify all of the customers from France, the UK, and the USA. The customer values are the column headings of the crosstab. The next fastest varying output is the selection of products, which has four values that specify types of products. The page dimensions are selections of two time values, which are the first and second calendar quarters of the year 2000, one channel value, which is the direct channel, and one promotion value, which is all promotions.

The `TransactionProvider` is `tp` and the `DataProvider` is `dp`. The `output` object is a `PrintWriter`.

Example 9–8 Navigating for a Crosstab View with Pages

```
// ...in someMethod...
Source salesAmountsForSelections = salesAmount.join(customerSel)
                                              .join(productSel);
                                              .join(timeSel);
                                              .join(channelSel);
                                              .join(promotionSel);
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a Cursor for salesAmountsForSelections
CursorManagerSpecification cursorMngrSpec =
     dp.createCursorManagerSpecification(salesAmountsForSelections);
SpecifiedCursorManager cursorMngr =
                             dp.createCursorManager(cursorMngrSpec);
Cursor salesForSelCursor = cursorMngr.createCursor();

// Send the Cursor to the printAsCrosstab method
printAsCrosstab(salesForSelCursor);

cursorMngr.close();
// ...the remainder of the code of someMethod...

// This method expects a CompoundCursor.
private void printAsCrosstab(Cursor cursor) {
  // Cast the Cursor to a CompoundCursor
  CompoundCursor rootCursor = (CompoundCursor) cursor;
  List outputs = rootCursor.getOutputs();
  int nOutputs = outputs.size();

  // Set the initial positions of all outputs
  Iterator outputIter = outputs.iterator();
  while (outputIter.hasNext())
    ((Cursor) outputIter.next()).setPosition(1);
```

```
  // The last output is fastest-varying; it represents columns.
  // The next to last output represents rows.
  // All other outputs are on the page.
  Cursor colCursor = (Cursor) outputs.get(nOutputs - 1);
  Cursor rowCursor = (Cursor) outputs.get(nOutputs - 2);
  ArrayList pageCursors = new ArrayList();
  for (int i = 0 ; i < nOutputs - 2 ; i++) {
    pageCursors.add(outputs.get(i));
  }

  // Get the base ValueCursor, which has the data values
  ValueCursor dataCursor = rootCursor.getValueCursor();

  // Print the pages of the crosstab
  printPages(pageCursors, 0, rowCursor, colCursor, dataCursor);
}

// Prints the pages of a crosstab
private void printPages(List pageCursors, int pageIndex, Cursor rowCursor,
                        Cursor colCursor, ValueCursor dataCursor) {
  // Get a Cursor for this page
  Cursor pageCursor = (Cursor) pageCursors.get(pageIndex);

  // Loop over the values of this page dimension
  do {
     // If this is the fastest-varying page dimension, print a page
    if (pageIndex == pageCursors.size() - 1) {
      // Print the values of the page dimensions
      printPageHeadings(pageCursors);

      // Print the column headings
      printColumnHeadings(colCursor);

      // Print the rows
      printRows(rowCursor, colCursor, dataCursor);
```

```
      // Print a couple of blank lines to delimit pages
      output.println();
      output.println();
    }

     // If this is not the fastest-varying page, recurse to the
     // next fastest varying dimension.
    else {
      printPages(pageCursors, pageIndex + 1, rowCursor, colCursor,
                 dataCursor);
    }
  } while (pageCursor.next());

  // Reset this page dimension Cursor to its first element.
  pageCursor.setPosition(1);
}

// Prints the values of the page dimensions on each page
private void printPageHeadings(List pageCursors) {
  // Print the values of the page dimensions
  Iterator pageIter = pageCursors.iterator();
  while (pageIter.hasNext())
    output.println(((ValueCursor) pageIter.next()).getCurrentValue());
  output.println();
}

// Prints the column headings on each page
private void printColumnHeadings(Cursor colCursor) {
  do {
     output.print("\t");
     output.print(((ValueCursor) colCursor).getCurrentValue());
  } while (colCursor.next());
  output.println();
  colCursor.setPosition(1);
}

// Prints the rows of each page
private void printRows(Cursor rowCursor, Cursor colCursor,
                       ValueCursor dataCursor) {
```

```
  // Loop over rows
  do {
    // Print row dimension value
    output.print(((ValueCursor) rowCursor).getCurrentValue());
    output.print("\t");
    // Loop over columns
    do {
      // Print data value
      output.print(dataCursor.getCurrentValue());
      output.print("\t");
    } while (colCursor.next());
    output.println();

    // Reset the column Cursor to its first element
    colCursor.setPosition(1);
  } while (rowCursor.next());

  // Reset the row Cursor to its first element
  rowCursor.setPosition(1);
}
```

The crosstab output of Example 9–8 looks like the following.

```
Promotion total
Direct
2000-Q1

                       FR           UK           US
Outerwear - Men        750563.50    938014.00    12773925.50
Outerwear - Women      984461.00    1388755.50   15421979.00
Outerwear - Boys       693382.00    799452.00    9183052.00
Outerwear - Girls      926520.50    977291.50    11854203.00

Promotion total
Direct
2000-Q2

                       FR            UK            US
Outerwear - Men        683521.00     711945.00     9947221.50
Outerwear - Women      840024.50     893587.50     12484221.00
Outerwear - Boys       600382.50     755031.00     8791240.00
Outerwear - Girls      901558.00     909421.50     9975927.00
```

Specifying the Behavior of a Cursor

You can specify the following aspects of the behavior of a `Cursor`.

- The *fetch size* of a `Cursor`, which is the number of elements of the result set that the `Cursor` retrieves during one fetch operation.
- The *shape* of the *fetch block* of a `Cursor`. The fetch block is the set of elements of each descendent `ValueCursor` that the parent `CompoundCursor` retrieves. The shape of the fetch block is the levels of the `CompoundCursor` at which you set the fetch sizes.
- Whether Oracle OLAP calculates the *extent* of the `Cursor`. The extent is the total number of positions of the `Cursor`. If the `Cursor` is a child `Cursor` of a `CompoundCursor`, its extent is relative to any slower varying outputs.
- Whether Oracle OLAP calculates the positions in the parent `Cursor` at which the value of a child `Cursor` starts or ends.

To specify the behavior of `Cursor`, you use methods on the `CursorSpecification` for that `Cursor`. To get the `CursorSpecification` for a `Cursor`, you use methods on the `CursorManagerSpecification` that you create for a `Source`.

Note: Specifying the calculation of the extent or the starting or ending position in a parent `Cursor` of the current value of a child `Cursor` can be a very expensive operation. The calculation can require considerable time and computing resources. You should only specify these calculations when your application needs them.

For more information on the relationships of `Source`, `Cursor`, `CursorSpecification`, and `CursorManagerSpecification` objects or the concepts of fetch size, extent, or `Cursor` positions, see Chapter 8.

Example 9–9 creates a `Source`, creates a `CursorManagerSpecification` for the `Source`, and then gets the `CursorSpecification` objects from a `CursorManagerSpecification`. The root `CursorSpecification` is the `CursorSpecification` for the top-level `CompoundCursor`.

Example 9–9 Getting CursorSpecification Objects from a CursorManagerSpecification

```
Source salesAmountsForSelections = salesAmount.join(customerSel)
                                              .join(productSel);
                                              .join(timeSel);
                                              .join(channelSel);
                                              .join(promotionSel);
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a Cursor for salesAmountsForSelections
CursorManagerSpecification cursorMngrSpec =
     dp.createCursorManagerSpecification(salesAmountsForSelections);

// Get the root CursorSpecification of the CursorManagerSpecification.
CompoundCursorSpecification rootCursorSpec =
(CompoundCursorSpecification) cursorMngrSpec.getRootCursorSpecification();

// Get the CursorSpecification for the base values
ValueCursorSpecification baseValueSpec =
                        rootCursorSpec.getValueCursorSpecification();

// Get the CursorSpecification objects for the outputs
List outputSpecs = rootCursorSpec.getOutputs();
ValueCursorSpecification promoSelValCSpec =
                       (ValueCursorSpecification) outputSpecs.get(0);
ValueCursorSpecification chanSelValCSpec =
                       (ValueCursorSpecification) outputSpecs.get(1);
ValueCursorSpecification timeSelValCSpec =
                       (ValueCursorSpecification) outputSpecs.get(2);
ValueCursorSpecification prodSelValCSpec =
                       (ValueCursorSpecification) outputSpecs.get(3);
ValueCursorSpecification custSelValCSpec =
                       (ValueCursorSpecification) outputSpecs.get(4);
```

Once you have the `CursorSpecification` objects, you can use their methods to specify the behavior of the `Cursor` objects that correspond to them.

Calculating Extent and Starting and Ending Positions of a Value

To manage the display of the result set retrieved by a `CompoundCursor`, you sometimes need to know the extent of its child `Cursor` components. You might also want to know the position at which the current value of a child `Cursor` starts in its parent `CompoundCursor`. You might want to know the *span* of the current value of a child `Cursor`. The span is the number of positions of the parent `Cursor` that the current value of the child `Cursor` occupies. You can calculate the span by subtracting the starting position of the value from its ending position and subtracting 1.

Before you can get the extent of a `Cursor` or get the starting or ending positions of a value in its parent `Cursor`, you must specify that you want Oracle OLAP to calculate the extent or those positions. To specify the performance of those calculations, you use methods on the `CursorSpecification` for the `Cursor`.

Example 9–10 specifies calculating the extent of a `Cursor`. The example uses the `CursorManagerSpecification` from Example 9–9.

Example 9–10 Specifying the Calculation of the Extent of a Cursor

```
CompoundCursorSpecification rootCursorSpec =
(CompoundCursorSpecification) cursorMngrSpec.getRootCursorSpecification();
rootCursorSpec.setExtentCalculationSpecified(true);
```

You can use methods on a `CursorSpecification` to determine whether the `CursorSpecification` specifies the calculation of the extent of a `Cursor` as in the following example.

```
boolean isSet = rootCursorSpec.isExtentCalculationSpecified();
```

Example 9–11 specifies calculating the starting and ending positions of the current value of a child `Cursor` in its parent `Cursor`. The example uses the `CursorManagerSpecification` from Example 9–9.

Example 9–11 Specifying the Calculation of Starting and Ending Positions in a Parent

```
CompoundCursorSpecification rootCursorSpec =
(CompoundCursorSpecification) cursorMngrSpec.getRootCursorSpecification();

// Get the List of CursorSpecification objects for the outputs.
// Iterate through the list, specifying the calculation of the extent
// for each output CursorSpecification.
Iterator iterOutputSpecs = rootCursorSpec.getOutputs().iterator();
ValueCursorSpecification valCursorSpec = (ValueCursorSpecification)
                                          iterOutputSpecs.next();
while(iterOutputSpecs.hasNext()) {
  valCursorSpec.setParentStartCalculationSpecified(true);
  valCursorSpec.setParentEndCalculationSpecified(true);
  valCursorSpec = (ValueCursorSpecification) iterOutputSpecs.next();
}
```

You can use methods on a `CursorSpecification` to determine whether the `CursorSpecification` specifies the calculation of the starting or ending positions of the current value of a child `Cursor` in its parent `Cursor`, as in the following example.

```
boolean isSet;
Iterator iterOutputSpecs = rootCursorSpec.getOutputs().iterator();
ValueCursorSpecification valCursorSpec = (ValueCursorSpecification)
                                          iterOutputSpecs.next();
while(iterOutputSpecs.hasNext()) {
  isSet = valCursorSpec.isParentStartCalculationSpecified();
  isSet = valCursorSpec.isParentEndCalculationSpecified();
  valCursorSpec = (ValueCursorSpecification) iterOutputSpecs.next();
}
```

Example 9–12 determines the span of the positions in a parent `CompoundCursor` of the current value of a child `Cursor` for two of the outputs of the `CompoundCursor`. The example uses the `salesAmountsForSelections` `Source` from Example 9–8.

The example gets the starting and ending positions of the current values of the time and product selections and then calculates the span of those values in the parent `Cursor`. The parent is the root `CompoundCursor`. The `TransactionProvider` is `tp`, the `DataProvider` is `dp`, and `output` is a `PrintWriter`.

Example 9–12 Calculating the Span of the Positions in the Parent of a Value

```
Source salesAmountsForSelections = salesAmount.join(customerSel)
                                              .join(productSel);
                                              .join(timeSel);
                                              .join(channelSel);
                                              .join(promotionSel);
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a CursorManagerSpecification for salesAmountsForSelections
CursorManagerSpecification cursorMngrSpec =
     dp.createCursorManagerSpecification(salesAmountsForSelections);

 // Get the root CursorSpecification from the CursorManagerSpecification.
CompoundCursorSpecification rootCursorSpec =
(CompoundCursorSpecification) cursorMngrSpec.getRootCursorSpecification();
// Get the CursorSpecification objects for the outputs
List outputSpecs = rootCursorSpec.getOutputs();
ValueCursorSpecification timeSelValCSpec =
 (ValueCursorSpecification) outputSpecs.get(2); \\ output for time
ValueCursorSpecification prodSelValCSpec =
 (ValueCursorSpecification) outputSpecs.get(3)  \\ output for product

// Specify the calculation of the starting and ending positions
timeSelValCSpec.setParentStartCalculationSpecified(true);
timeSelValCSpec.setParentEndCalculationSpecified(true);
prodSelValCSpec.setParentStartCalculationSpecified(true);
prodSelValCSpec.setParentEndCalculationSpecified(true);

// Create the CursorManager and the Cursor
SpecifiedCursorManager cursorMngr = dp.createCursorManager(cursorMngrSpec);
CompoundCursor cursor = (CompoundCursor) cursorMngr.createCursor();
```

```
// Get the child Cursor objects
ValueCursor baseValCursor = cursor.getValueCursor();
List outputs = cursor.getOutputs();
ValueCursor promoSelVals = (ValueCursor) outputs.get(0);
ValueCursor chanSelVals = (ValueCursor) outputs.get(1);
ValueCursor timeSelVals = (ValueCursor) outputs.get(2);
ValueCursor custSelVals = (ValueCursor) outputs.get(3);
ValueCursor prodSelVals = (ValueCursor) outputs.get(4);

// Set the position of the root CompoundCursor
cursor.setPosition(15);
/*
 * Get the values at the current position and determine the span
 * of the values of the time and product outputs.
 */
output.print(promoSelVals.getCurrentValue() + ", ");
output.print(chanSelVals.getCurrentValue() + ", ");
output.print(timeSelVals.getCurrentValue() + ", ");
output.print(custSelVals.getCurrentValue() + ", ");
output.print(prodSelVals.getCurrentValue() + ", ");
output.println(baseValCursor.getCurrentValue());

// Determine the span of the values of the two fastest varying outputs
int span;
span = (prodSelVals.getParentEnd() - prodSelVals.getParentStart()) -1);
output.println("The span of " + prodSelVals.getCurrentValue() +
" at the current position is " + span + ".")
span = (timeSelVals.getParentEnd() - timeSelVals.getParentStart()) -1);
output.println("The span of " + timeSelVals.getCurrentValue() +
" at the current position is " + span + ".")
cursorMngr.close();
```

This example produces the following output.

```
Promotion total, Direct, 2000-Q1, Outerwear - Men, US, 9947221.50
The span of Outerwear - Men at the current position is 3.
The span of 2000-Q2 at the current position is 12.
```

Specifying Fetch Sizes and Fetch Blocks

The number of elements of a `Cursor` that Oracle OLAP sends to the client application during one fetch operation depends on the fetch size specified for that `Cursor`. For a `CompoundCursor`, you can set the fetch size on the `CompoundCursor` itself or at one or more levels of its descendent `Cursor`

components. Setting the fetch size on a `CompoundCursor` specifies that fetch size for its child `Cursor` components.

The set of elements the `Cursor` retrieves in a single fetch is the fetch block. The shape of the fetch block is determined by the set of `Cursor` components on which you set the fetch sizes. For more information on fetch sizes and fetch blocks, see Chapter 8.

You specify the shape of the fetch block and the specific fetch sizes according to the needs of your display of the data. To display the results of a query in a table view, you specify the fetch size on the top-level `CompoundCursor`.

To display the results in a crosstab view, you specify the fetch sizes on the children of the top-level `CompoundCursor`. For a crosstab that displays the results of a query that has nested levels of outputs, you might specify fetch sizes at different levels of the children of the component `CompoundCursor` objects.

You use methods on a `CursorSpecification` to set the default fetch size for its `Cursor`. For a `CompoundCursorSpecification`, you can specify setting the fetch sizes on its children and thereby determine the shape of the fetch block.

If a default fetch size is set on a `CursorSpecification`, you can use the `setFetchSize` method on the `Cursor` for that `CursorSpecification` to change the fetch size of the `Cursor`. By default, the root `CursorSpecification` of a `CursorManagerSpecification` has the fetch size set to 100.

Example 9–13 creates a `Source` that represents the sales amount measure values as specified by selections of values from the dimensions of the measure. The product and customer selections each have ten values, the time selection has four values, and the promotion and channel selections each have one value. Assuming that a sales amount exists for each set of dimension values, the result set of the query has 300 elements (10*10*3*1*1).

To match a display of the elements that contains only twenty rows, the example sets a fetch size of twenty elements on the top-level `CompoundCursor`. Because the default fetch size is automatically set on the root `CursorSpecification`, which in this example is the `CompoundCursorSpecification` for the top-level `CompoundCursor`, the example just uses the `setFetchSize` method on the `CompoundCursor` to change the fetch size. The fetch block is the set of output and base values specified by twenty positions of the top-level `CompoundCursor`. The `TransactionProvider` is `tp` and the `DataProvider` is `dp`.

Example 9–13 Specifying the Fetch Size and Fetch Block for a Table View

```
Source salesAmountsForSelections = salesAmount.join(customerSel)
                                              .join(productSel);
                                              .join(timeSel);
                                              .join(channelSel);
                                              .join(promotionSel);
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a Cursor for salesAmountsForSelections
CursorManagerSpecification cursorMngrSpec =
     dp.createCursorManagerSpecification(salesAmountsForSelections);
SpecifiedCursorManager cursorMngr = dp.createCursorManager(cursorMngrSpec);
Cursor cursor = cursorMngr.createCursor();

// Set the fetch size of the top-level CompoundCursor to 20
cursor.setFetchSize(20);
```

Example 9–14 modifies the example in Example 9–7. In Example 9–14, the number of times that the `for` loops are repeated depends upon the extent of the `Cursor`. As the conditional statement of the `for` loops, instead of specifying the number of positions that the `Cursor` has, this example gets the extent of the `Cursor` and uses the extent as the condition. The optimal fetch block for the crosstab display is a fetch block that contains, for each position of the `CompoundCursor`, the extent of the child `Cursor` elements at that position.

This example creates a `CursorManagerSpecification` and gets the root `CursorSpecification`. It casts the root `CursorSpecification` as a `CompoundCursorSpecification`. The example specifies setting the default fetch sizes on the children of the root `CompoundCursorSpecification` and it specifies the calculation of its extent.

The example sets the fetch size on each output `ValueCursor` equal to the extent of the `ValueCursor`. It then gets the displayable portion of the crosstab by looping through the positions of the child `ValueCursor` objects.

Example 9–14 Using Extents To Specify the Fetch Sizes for a Crosstab View

```
Source unitPriceByDay = unitPrice.join(productSel)
                                 .join(timeSel);
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a CursorManagerSpecification for unitPriceByDay
CursorManagerSpecification cursorMngrSpec =
            dp.createCursorManagerSpecification(unitPriceByDay);

// Get the root CursorSpecification and cast it to a
// CompoundCursorSpecification
CompoundCursorSpecification rootSpec =
(CompoundCursorSpecification) cursorMngrSpec.getRootCursorSpecification();

// Specify setting the fetch size on the child Cursor objects
// and calculating the extent of the positions in the Cursor
rootSpec.specifyDefaultFetchSizeOnChildren();
rootSpec.setExtentCalculationSpecified(true);

// Create the CursorManager and the Cursor
SpecifiedCursorManager cursorMngr =
                             dp.createCursorManager(cursorMngrSpec);
Cursor unitPriceByDayCursor = cursorMngr.createCursor();

// Cast the Cursor to a CompoundCursor
CompoundCursor rootCursor = (CompoundCursor) unitPriceByDayCursor;

// Determine a starting position and the number of rows to display.
// The position in columnCursor at which the current display starts
// is colStart and rowStart is the position in rowCursor at which
// the current display starts.
int colStart = 1;
int rowStart = 1;
String productValue;
String timeValue;
double price;
```

```
// The number of values from the ValueCursor objects for products and
// days are now initialized as 1 because the ValueCursor objects have
// at least one element.
int numProducts = 1;
int numDays = 1;

// Get the ValueCursor and the outputs
CompoundCursor rootCursor = (CompoundCursor) unitPriceByDayCursor;
List outputs = rootCursor.getOutputs();
// The first output has the values of timeSel, the slower varying output
ValueCursor rowCursor = (ValueCursor) outputs.get(0);
// The second output has the faster varying values of productSel
ValueCursor columnCursor = (ValueCursor) outputs.get(1);
ValueCursor unitPriceValues = rootCursor.getValueCursor();// Prices

// Loop through the positions of the faster varying output Cursor
for(int pPos = colStart; pPos < colStart + numProducts; pPos++) {
  columnCursor.setPosition(pPos);
  // Get the extents of the output ValueCursor objects
  numProducts = columnCursor.getExtent();
  numDays = rowCursor.getExtent();
  // Set the fetch sizes
  columnCursor.setFetchSize(numProducts);
  rowCursor.setFetchSize(numMonths);
  // Loop through the positions of the slower varying output Cursor
  for(int tPos = rowStart; tPos < rowStart + numDays; tPos++) {
    rowCursor.setPosition(tPos);

    // Get the values. Sending the values to the appropriate
    // display mechanism is not shown.
    productValue = columnCursor.getCurrentString();
    timeValue = rowCursor.getCurrentString();
    price = unitPriceValues.getCurrentDouble();
  }
}
```

10

Creating Dynamic Queries

This chapter describes the Oracle OLAP API `Template` class and its related classes, which you use to create dynamic queries. This chapter also provides examples of implementations of those classes.

This chapter includes the following topics:

- About Template Objects
- Overview of Template and Related Classes
- Designing and Implementing a Template

About Template Objects

The Template class is the basis of a very powerful feature of the Oracle OLAP API. You use Template objects to create modifiable Source objects. With those Source objects, you can create dynamic queries that can change in response to end-user selections. Template objects also offer a convenient way for you to translate user-interface elements into OLAP API operations and objects.

These features are briefly described below. The rest of this chapter describes the Template class and the other classes you use to create dynamic Source objects. For information on the Transaction objects that you use to make changes to the dynamic Source and to either save or discard those changes, see Chapter 7.

About Creating a Dynamic Source

The main feature of a Template is its ability to produce a dynamic Source. That ability is based on two of the other objects that a Template uses: instances of the DynamicDefinition and MetadataState classes.

When a Source is created, a SourceDefinition is automatically created. The SourceDefinition has information about how the Source was created. Once created, the Source and its SourceDefinition are paired immutably. The getSource method of a SourceDefinition gets its paired Source.

DynamicDefinition is a subclass of SourceDefinition. A Template creates a DynamicDefinition, which acts as a proxy for the SourceDefinition of the Source produced by the Template. This means that instead of always getting the same immutably paired Source, the getSource method on the DynamicDefinition gets whatever Source is currently produced by the Template. The instance of the DynamicDefinition does not change even though the Source that it gets is different.

The Source that a Template produces can change because the values, including other Source objects, that the Template uses to create the Source can change. A Template stores those values in a MetadataState. A Template provides methods to get the current state of the MetadataState, to get or set a value, and to set the state. You use those methods to change the data values the MetadataState stores.

You use a DynamicDefinition to get the Source produced by a Template. If your application changes the state of the values that the Template uses to create the Source, for example, in response to end-user selections, then the application uses the same DynamicDefinition to get the Source again, even though the new Source defines a result set different than the previous Source.

The Source produced by a Template can be the result of a series of Source operations that create other Source objects, such as a series of selections, sorts, calculations, and joins. You put the code for those operations in the generateSource method of a SourceGenerator for the Template. That method returns the Source produced by the Template. The operations use the data stored in the MetadataState.

You might build an extremely complex query that involves the interactions of dynamic Source objects produced by many different Template objects. The end result of the query building is a Source that defines the entire complex query. If you change the state of any one of the Template objects that you used to create the final Source, then the final Source represents a result set different than that of the previous Source. You can thereby modify the final query without having to reproduce all of the operations involved in defining the query.

About Translating User Interface Elements into OLAP API Objects

You design Template objects to represent elements of the user interface of an application. Your Template objects turn the selections that the end user makes into OLAP API query-building operations that produce a Source. You then create a Cursor to fetch the result set defined by the Source from Oracle OLAP. You get the values from the Cursor and display them to the end user. When an end user makes changes to the selections, you change the state of the Template. You then get the Source produced by the Template, create a new Cursor, get the new values, and display them.

Overview of Template and Related Classes

In the OLAP API, several classes work together to produce a dynamic Source. In designing a Template, you must implement or extend the following:

- The Template abstract class
- The MetadataState interface
- The SourceGenerator interface

Instances of those three classes, plus instances of other classes that Oracle OLAP creates, work together to produce the Source that the Template defines. The

classes that Oracle OLAP provides, which you create by calling factory methods, are the following:

- `DataProvider`
- `DynamicDefinition`

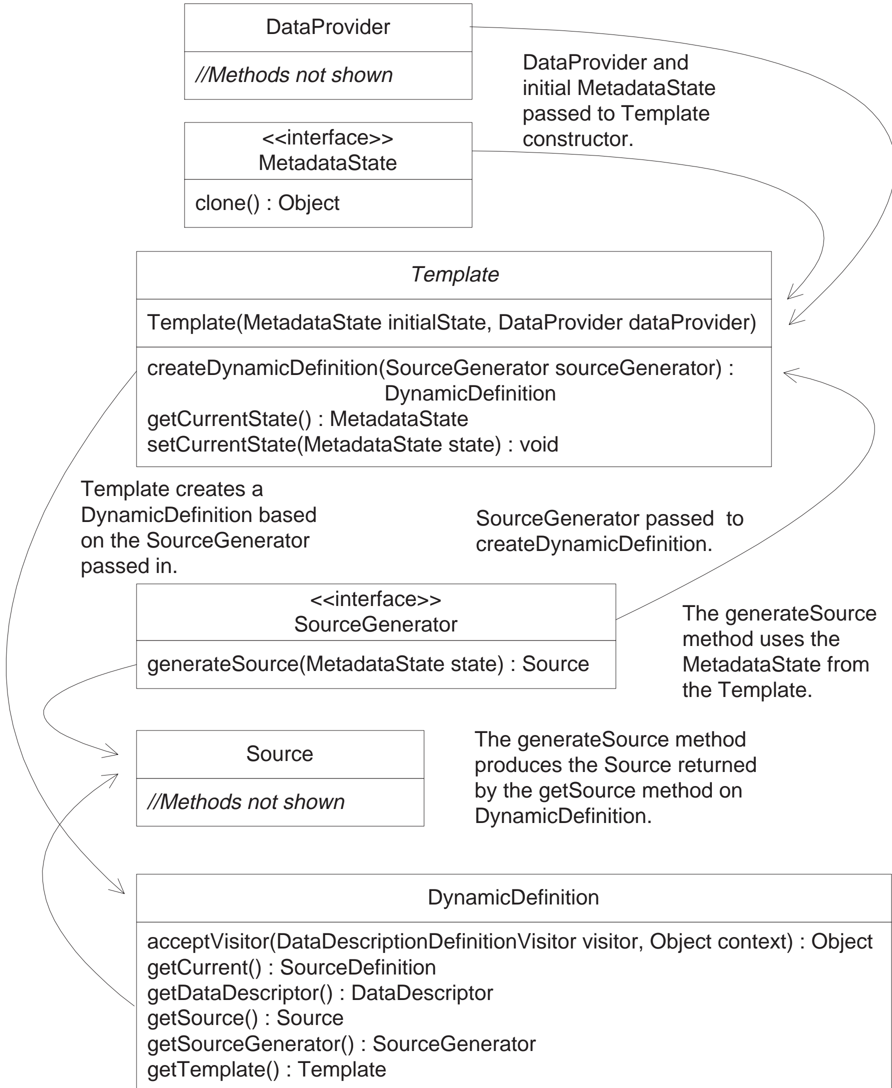
What Is the Relationship Between the Classes That Produce a Dynamic Source?

The classes that produce a dynamic `Source` work together as follows:

- A `Template` has methods that create a `DynamicDefinition` and that get and set the current state of a `MetadataState`. An extension to the `Template` abstract class adds methods that get and set the values of fields on the `MetadataState`.
- The `MetadataState` implementation has fields for storing the data to use in generating the `Source` for the `Template`. When you create a new `Template`, you pass the `MetadataState` to the constructor of the `Template`. When you call the `getSource` method on the `DynamicDefinition`, the `MetadataState` is passed to the `generateSource` method on the `SourceGenerator`.
- The `DataProvider` is used in creating a `Template` and by the `SourceGenerator` in creating new `Source` objects.
- The `SourceGenerator` implementation has a `generateSource` method that uses the current state of the data in the `MetadataState` to produce a `Source` for the `Template`. You pass in the `SourceGenerator` to the `createDynamicDefinition` method on the `Template` to create a `DynamicDefinition`.
- The `DynamicDefinition` has a `getSource` method that gets the `Source` produced by the `SourceGenerator`. The `DynamicDefinition` serves as a proxy for the immutably paired `SourceDefinition` of that `Source`.

Figure 10–1 illustrates the relationship of the classes described in the preceding list. The arrows on the right indicate that the `DataProvider` and `MetadataState` objects are passed to the `Template` constructor and that the `SourceGenerator` is passed to the `createDynamicDefinition` method on the `Template`. The arrows on the left indicate that a `DynamicDefinition` is returned by the `createDynamicDefinition` method and that the same `Source` is returned by the `generateSource` method on the `SourceGenerator` and the `getSource` method on the `DynamicDefinition`.

Figure 10–1 The Relationship of the Classes That Produce a Dynamic Source



Template Class

You use a Template to produce a modifiable Source. A Template has methods for creating a DynamicDefinition and for getting and setting the current state of the Template. In extending the Template class, you add methods that provide access to the fields on the MetadataState for the Template. The Template creates a DynamicDefinition that you use to get the Source produced by the SourceGenerator for the Template.

For an example of a Template implementation, see Example 10–1 on page 10-9.

MetadataState Interface

An implementation of the MetadataState interface stores the current state of the values for a Template. A MetadataState must include a clone method that creates a copy of the current state.

When instantiating a new Template, you pass a MetadataState to the Template constructor. The Template has methods for getting and setting the values stored by the MetadataState. The generateSource method on the SourceGenerator for the Template uses the MetadataState when the method produces a Source for the Template.

For an example of a MetadataState implementation, see Example 10–2 on page 10-12.

SourceGenerator Interface

An implementation of SourceGenerator must include a generateSource method, which produces a Source for a Template. A SourceGenerator must produce only one type of Source, such as a BooleanSource, a NumberSource, or a StringSource. In producing the Source, the generateSource method uses the current state of the data represented by the MetadataState for the Template.

To get the Source produced by the generateSource method, you create a DynamicDefinition by passing the SourceGenerator to the createDynamicDefinition method on the Template. You then get the Source by calling the getSource method on the DynamicDefinition.

A Template can create more than one DynamicDefinition, each with a differently implemented SourceGenerator. The generateSource methods on the different SourceGenerator objects use the same data, as defined by the

current state of the MetadataState for the Template, to produce Source objects that define different queries.

For an example of a SourceGenerator implementation, see Example 10–3 on page 10-13.

DynamicDefinition Class

DynamicDefinition is a subclass of SourceDefinition. You create a DynamicDefinition by calling the createDynamicDefinition method on a Template and passing it a SourceGenerator. You get the Source produced by the SourceGenerator by calling the getSource method on the DynamicDefinition.

A DynamicDefinition created by a Template is a proxy for the SourceDefinition of the Source produced by the SourceGenerator. The SourceDefinition is immutably paired to its Source. If the state of the Template changes, then the Source produced by the SourceGenerator is different. Because the DynamicDefinition is a proxy, you use the same DynamicDefinition to get the new Source even though that Source has a different SourceDefinition.

The getCurrent method of a DynamicDefinition returns the SourceDefinition immutably paired to the Source that the generateSource method currently returns. For an example of the use of a DynamicDefinition, see Example 10–4 on page 10-15.

Designing and Implementing a Template

The design of a Template reflects the query-building elements of the user interface of an application. For example, suppose you want to develop an application that allows the end user to create a query that requests a number of values from the top or bottom of a list of values. The values are from one dimension of a measure. The other dimensions of the measure are limited to single values.

The user interface of your application has a dialog box that allows the end user to do the following:

- Select a radio button that specifies whether the data values should be from the top or bottom of the range of values.
- Select a measure from a drop-down list of measures.
- Select a number from a field. The number specifies the number of data values to display.
- Select one of the dimensions of the measure as the base of the data values to display. For example, if the user selects the product dimension, then the query specifies some number of products from the top or bottom of the list of products. The list is determined by the measure and the selected values of the other dimensions.
- Click a button to bring up a Single Selections dialog box through which the end user selects the single values for the other dimensions of the selected measure. After selecting the values of the dimensions, the end user clicks an OK button on the second dialog box and returns to the first dialog box.
- Click an OK button to generate the query. The results of the query appear.

To generate a `Source` that represents the query that the end user creates in the first dialog box, you design a `Template` called `TopBottomTemplate`. You also design a second `Template`, called `SingleSelectionTemplate`, to create a `Source` that represents the end user's selections of single values for the dimensions other than the base dimension. The designs of your `Template` objects reflect the user interface elements of the dialog boxes.

In designing the `TopBottomTemplate` and its `MetadataState` and `SourceGenerator`, you do the following:

- Create a class called `TopBottomTemplate` that extends `Template`. To the class, you add methods that get the current state of the `Template`, set the values specified by the user, and then set the current state of the `Template`.
- Create a class called `TopBottomTemplateState` that implements `MetadataState`. You provide fields on the class to store values for the `SourceGenerator` to use in generating the `Source` produced by the `Template`. The values are set by methods of the `TopBottomTemplate`.
- Create a class called `TopBottomTemplateGenerator` that implements `SourceGenerator`. In the `generateSource` method of the class, you provide the operations that create the `Source` specified by the end user's selections.

Using your application, an end user selects sales amount as the measure and products as the base dimension in the first dialog box. From the Single Selections dialog box, the end user selects customers from San Francisco, the first quarter of 2000, the direct channel, and billboard promotions as the single values for each of the remaining dimensions.

The query that the end user has created requests the ten products that have the highest total sales amount values of those sold through the direct sales channel to customers from San Francisco during the first calendar quarter of the year 2000 while a billboard promotion was occurring.

For examples of implementations of the `TopBottomTemplate`, `TopBottomTemplateState`, and `TopBottomTemplateGenerator` objects, and an example of an application that uses them, see Example 10–1, Example 10–2, Example 10–3, and Example 10–4.

Implementing the Classes for a Template

Example 10–1 is an implementation of the `TopBottomTemplate` class.

Example 10–1 Implementing a Template

```
package myTestPackage;

import oracle.olapi.data.source.DataProvider;
import oracle.olapi.data.source.DynamicDefinition;
import oracle.olapi.data.source.Source;
import oracle.olapi.data.source.Template;
import oracle.olapi.transaction.metadataStateManager.MetadataState;

/**
 * Creates a TopBottomTemplateState, a TopBottomTemplateGenerator,
 * and a DynamicDefinition. Gets the current state of the
 * TopBottomTemplateState and the values it stores. Sets the data values
 * stored by the TopBottomTemplateState and sets the changed state as
 * the current state.
 */
public class TopBottomTemplate extends Template {
  public static final int TOP_BOTTOM_TYPE_TOP = 0;
  public static final int TOP_BOTTOM_TYPE_BOTTOM = 1;

  // Variable to store the DynamicDefinition.
  private DynamicDefinition _definition;
```

```
/**
 * Creates a TopBottomTemplate with default type and number values
 * and a specified base dimension.
 */
public TopBottomTemplate(Source base, DataProvider dataProvider) {
  super(new TopBottomTemplateState(base, TOP_BOTTOM_TYPE_TOP, 0),
                                   dataProvider);
  // Create the DynamicDefinition for this Template. Create the
  // TopBottomTemplateGenerator that the DynamicDefinition uses.
  _definition =
  createDynamicDefinition(new TopBottomTemplateGenerator(dataProvider));
}

/**
 * Gets the Source produced by the TopBottomTemplateGenerator
 * from the DynamicDefinition.
 */
public final Source getSource() {
  return _definition.getSource();
}

/**
 * Gets the Source that is the base of the values in the result set.
 * Returns null if the state has no base.
 */
public Source getBase() {
  TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
  return state.base;
}

/**
 * Sets a Source as the base.
 */
public void setBase(Source base) {
  TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
  state.base = base;
  setCurrentState(state);
}
```

```
/**
 * Gets the Source that specifies the measure and the single
 * selections from the dimensions other than the base.
 */
public Source getCriterion() {
  TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
  return state.criterion;
}

/**
 * Specifies a Source that defines the measure and the single values
 * selected from the dimensions other than the base.
 * The SingleSelectionTemplate produces such a Source.
 */
public void setCriterion(Source criterion) {
  TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
  state.criterion = criterion;
  setCurrentState(state);
}

/**
 * Gets the type, which is either TOP_BOTTOM_TYPE_TOP or
 * TOP_BOTTOM_TYPE_BOTTOM.
 */
public int getTopBottomType() {
  TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
  return state.topBottomType;
}

/**
 * Sets the type.
 */
public void setTopBottomType(int topBottomType) {
  if ((topBottomType < TOP_BOTTOM_TYPE_TOP) ||
      (topBottomType > TOP_BOTTOM_TYPE_BOTTOM))
    throw new IllegalArgumentException("InvalidTopBottomType");
  TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
  state.topBottomType = topBottomType;
  setCurrentState(state);
}
```

```
  /**
   * Gets the number of values selected.
   */
  public float getN() {
    TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
    return state.N;
  }

  /**
   * Sets the number of values to select.
   */
  public void setN(float N) {
    TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
    state.N = N;
    setCurrentState(state);
  }
}
```

Example 10–2 is an implementation of the `TopBottomTemplateState` class described earlier.

Example 10–2 Implementing a MetadataState

```
package myTestPackage;

import oracle.olapi.data.source.Source;
import oracle.olapi.transaction.metadataStateManager.MetadataState;

/**
 * Stores data that can be changed by its TopBottomTemplate.
 * The data is used by a TopBottomTemplateGenerator in producing
 * a Source for the TopBottomTemplate.
 */
public final class TopBottomTemplateState
     implements Cloneable, MetadataState {
  public int topBottomType;
  public float N;
  public Source criterion;
  public Source base;
```

```
  /**
   * Creates a TopBottomTemplateState.
   */
  public TopBottomTemplateState(Source base, int topBottomType, float N) {
    this.base = base;
    this.topBottomType = topBottomType;
    this.N = N;
  }

  /**
   * Creates a copy of this TopBottomTemplateState.
   */
  public final Object clone() {
    try {
      return super.clone();
    }
    catch(CloneNotSupportedException e) {
      return null;
    }
  }
}
```

Example 10–3 is an implementation of the `TopBottomTemplateGenerator` class described earlier.

Example 10–3 Implementing a SourceGenerator

```
package myTestPackage;

import oracle.olapi.data.source.DataProvider;
import oracle.olapi.data.source.Source;
import oracle.olapi.data.source.SourceGenerator;
import java.lang.Math;

/**
 * Produces a Source for a TopBottomTemplate based on the data
 * values of a TopBottomTemplateState.
 */
public final class TopBottomTemplateGenerator
      implements SourceGenerator {
  // Store the DataProvider.
  private DataProvider _dataProvider;
```

```
  /**
   * Creates a TopBottomTemplateGenerator.
   */
  public TopBottomTemplateGenerator(DataProvider dataProvider) {
    _dataProvider = dataProvider;
  }

  /**
   * Generates a Source for a TopBottomTemplate using the current
   * state of the data values stored by the TopBottomTemplateState.
   */
  public Source generateSource(MetadataState state) {
    TopBottomTemplateState castState = (TopBottomTemplateState)state;
    if (castState.criterion == null)
      throw new NullPointerException("CriterionParameterMissing");
    Source sortedBase = null;
    if (castState.topBottomType == TOP_BOTTOM_TYPE_TOP)
      sortedBase = castState.base.sortDescending(castState.criterion);
    else
      sortedBase = castState.base.sortAscending(castState.criterion);
    return sortedBase.interval(1, Math.round(castState.N));
  }
}
```

Implementing an Application That Uses Templates

After you have stored the selections made by the end user in the `MetadataState` for the `Template`, use the `getSource` method on the `DynamicDefinition` to get the `Source` created by the `Template`. This section provides an example of an application that uses the `TopBottomTemplate` described in Example 10–1. For brevity, the code does not contain much exception handling.

The `Context` class used in the example has methods that do the following:

- Connects to Oracle OLAP.
- Opens a database.
- Gets metadata objects for the measure and the dimensions selected by the end user.
- Gets primary `Source` objects from the metadata objects.

The example does the following:

- Gets primary `Source` objects from the `Context`.
- Creates a `SingleSelectionTemplate` for selecting single values from some of the dimensions of the measure.
- Creates a `TopBottomTemplate` and stores selections made by the end user.
- Gets the `Source` produced by the `TopBottomTemplate`.
- Creates a `Cursor` for that `Source`.
- Gets the values from the `Cursor` and displays them.

Example 10–4 does not include the code for interacting with the end user or for implementing the `SingleSelectionTemplate` or the `MetadataState` and `SourceGenerator` objects for the `SingleSelectionTemplate`. The example class has a method for creating a `Cursor` and a method for printing the values of the `Cursor`. All other operations occur in the `main` method. The `Context` object supplies the connection to the database, the `DataProvider` and the `TransactionProvider`, and primary `Source` objects.

Example 10–4 Getting the Source Produced by the Template

```
package myTestPackage;

import oracle.olapi.data.source.Source;
import oracle.olapi.data.source.StringSource;
import oracle.olapi.data.source.DataProvider;
import oracle.olapi.data.source.CursorManagerSpecification;
import oracle.olapi.data.cursor.CursorManager;
import oracle.olapi.data.source.SpecifiedCursorManager;
import oracle.olapi.data.cursor.Cursor;
import oracle.olapi.data.cursor.ValueCursor;
import oracle.olapi.transaction.NotCommittableException;
import myTestPackage.Context;
import myTestPackage.TopBottomTemplate;
import myTestPackage.SingleSelectionTemplate;
```

```
/**
 * Creates a query that specifies a number of values from the top or
 * bottom of a list of values from one of the dimensions of a measure.
 * The list is determined by the measure and by single values from
 * the other dimensions of the measure. Displays the results of the
 * query.
 */
public class TopBottomTest {
  /**
   * Prints the values of the Cursor.
   */
  public static void printCursor(Cursor cursor) {

    // Because the result is a single set of values with no outputs,
    // cast the Cursor to a ValueCursor and print out the values.
    ValueCursor valueCursor = (ValueCursor) cursor;
    int i = 1;
    do {
      System.out.println(i + ". " + valueCursor.getCurrentValue());
      i++;
    } while(valueCursor.next());
  }

  /**
   * Creates a Cursor.
   */
  public static void createCursor(Source choice, DataProvider dp) {
    CursorManagerSpecification cursorMngrSpec =
                      dp.createCursorManagerSpecification(choice);
    SpecifiedCursorManager cursorManager =
                      dp.createCursorManager(cursorMngrSpec);
    Cursor cursor = cursorManager.createCursor();
    // Print the values of the Cursor.
    printCursor(cursor);
    // Close the CursorManager.
    cursorManager.close();
  }
```

```
public static void main(String[] args) {

  // Create a Context object and from it get the DataProvider and
  // the primary Source objects for the measure and the dimensions.
  Context context = new Context();
  DataProvider dp = context.getDataProvider();
  Source[] sources = context.getPrimarySourcesByName(
        new String[]{"SALES_AMOUNT", "PRODUCTS_DIM", "CUSTOMERS_DIM",
                     "CHANNELS_DIM", "TIMES_DIM", "PROMOTIONS_DIM"});
  Source salesAmount = sources[0];
  StringSource product = (StringSource)sources[1];
  StringSource customer = (StringSource)sources[2];
  StringSource channel = (StringSource)sources[3];
  StringSource time = (StringSource)sources[4];
  StringSource promo = (StringSource)sources[5];

  // Create a SingleSelectionTemplate to produce a Source that
  // specifies a single value for each of the dimensions other
  // than the base for the selected measure.
  SingleSelectionTemplate singleSelections =
                      new SingleSelectionTemplate(salesAmount, dp);
  singleSelections.addSelection((StringSource) customer,
                                              "San Francisco");
  singleSelections.addSelection((StringSource) time, "2000-Q1");
  // S is the direct sales channel
  singleSelections.addSelection((StringSource) channel, "S");
  singleSelections.addSelection((StringSource) promo, "billboard");

  // Create a TopBottomTemplate and set the parameters selected by
  // the end user, including a dimension as the base and the
  // Source produced by the SingleSelectionTemplate as the
  // criterion.
  TopBottomTemplate topNBottom = new TopBottomTemplate(product, dp);
  topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_TOP);
  topNBottom.setN(15);
  topNBottom.setCriterion(singleSelections.getSource());
```

```
        // With methods on the TransactionProvider, prepare and commit
        // the transaction.
        try{
          context.getTransactionProvider().prepareCurrentTransaction();
        }
        catch(NotCommittableException e){
          System.out.println("Cannot prepare current Transaction. " +
                             "Caught exception " + e + ".");
        }
        context.getTransactionProvider().commitCurrentTransaction();

        // Get the Source produced by the TopBottomTemplate,
        // create a Cursor for it and display the results.
        createCursor(topNBottom.getSource(), dp);
      }
    }
```

A

Setting Up the Development Environment

This appendix describes the development environment for creating applications that use the OLAP API.

This appendix includes the following topics:

- Overview
- Required Software
- Setting Up on Your Application Development Computer
- Considerations for Deploying Your Application

Overview

The Oracle installation, with the OLAP option, provides all of the Oracle OLAP software that is required in the database and on its host computer. In addition, the Client installation provides `jar` files that are needed on the application development computer for creating an OLAP API client application.

As an application developer, you must complete the Client installation with the Administrator option, which copies these `jar` files to the computer on which you will write your Java application. In addition, you must ensure that supporting JDBC and Java files are available on the development computer.

Required Software

The application development computer must have the following files:

- OLAP API `jar` files, which represent the OLAP API client software. The Oracle Client installation with the Administrator option provides these files, along with the OLAP API Javadoc.
- Oracle JDBC (Java Database Connectivity) `jar` files, which provide communications between the application and the Oracle database. The Oracle Client installation with the Administrator option provides JDBC. For additional information about using the Oracle implementation of JDBC, see the Oracle Technology Web site at

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

  You must use Oracle's implementation, not a product from another vender.
- The Java Development Kit (JDK) version 1.2. The Oracle installation does not provide the JDK. For information about obtaining and using it, see the Sun Microsystems Java Web site at

  `http://java.sun.com`

If you are using Oracle JDeveloper as your development environment, JDBC and the JDK are already installed on your computer. However, ensure that you are using the correct version of the JDK in JDeveloper.

Setting Up on Your Application Development Computer

Installing the jar files

To make the `jar` files accessible in your development environment, take the following steps:

1. On your application development computer, start the Oracle Client Software CD for your platform.
2. Select the Administrator installation type, and complete the installation as directed.
3. Find the OLAP API `jar` files on your computer where the installation procedure copied them. Look in the `olap/olapi/lib` subdirectory of the Oracle home directory.
4. Make the OLAP API `jar` files accessible to the Java integrated development environment (IDE) that you are using. An example of an IDE is Oracle JDeveloper.
5. Edit your Java `CLASSPATH` environment variable to include the paths of the files on your computer.
6. In the IDE, make any specifications that are required to make the files accessible for importing classes into your programs.

Installing the OLAP API Javadoc

If you want to access the OLAP API Javadoc on your application development computer, locate the `jar` files that contain them on your computer where the installation procedure copied them. Look in the `olap/olapi/doc` subdirectory of the Oracle home directory. Consult the `readme.txt` file in that directory for instructions on how to install the files and access them in your Web browser.

Using a Sample Program

If you want to examine or run a sample Java program that uses the OLAP API, you can obtain it from the Oracle MetaLink (iSupport) Web site. To get the program, list the patches for the Oracle OLAP product in the Oracle Server product family on the Solaris or Windows NT platform. Search for a patch named "OLAP API Sample Program," and download the file.

Considerations for Deploying Your Application

When you deploy your application, ensure that the following are installed on each computer that will run the OLAP API:

- The OLAP API `jar` files
- Oracle JDBC
- A Java Runtime Environment (JRE)

For JDBC and the JRE, ensure that the installed version is compatible with the version that you used when you developed your application.

Index

A

aggregation functions, creating, 6-29
aggregation methods
  explanation of, 6-24
  list of, 6-25
  using, 6-25, 6-26
alias method
  description, 5-7
  example, 6-16
ancestors attributes
  example of getting, 4-8
  for MdmHierarchy objects, 2-12
  for MdmLevel objects, 2-10
application
  deployment, 1-12
  development steps, 1-8
  tasks performed by, 1-12
asymmetric result set, Cursor positions in an, 8-20
attributes
  ancestors, 2-10, 2-12
  definition, 1-3
  in OLAP metadata, 2-3
  MdmAttribute objects, 2-23
  parent, 2-10, 2-12
  region, 2-12
  Source objects for, 5-7

B

Boolean OLAP API data type, 2-25, 5-9, 5-10

C

CompoundCursor objects
  getting children of, example, 9-5
  navigating for a crosstab view, example, 9-12, 9-14
  navigating for a table view, example, 9-10
  positions of, 8-17
Connection objects
  example of closing, 3-5
  example of creating, 3-2
  example of getting an existing one, 3-4
connections
  closing, 3-5
  getting existing ones, 3-4
  prerequisites, 3-2
  steps for establishing, 3-2
constant Source objects
  definition, 5-5
  example, 5-8
crosstab view
  navigating Cursor for, example, 9-12, 9-14
current position in a Cursor, definition, 8-16
Cursor class
  architecture, advantages of, 8-2
  hierarchy, 8-4

Cursor objects
created in the current Transaction, 8-4
creating, example, 9-2
current position, definition, 8-16
extent calculation, example, 9-21
extent definition, 8-25
faster and slower varying components, 8-6
fetch block definition, 8-27
fetch size definition, 8-27
getting children of, example, 9-5
getting the values of, examples, 9-3
parent starting and ending position, 8-22
position, 8-16
Source objects for which you cannot create, 8-3
span, definition, 8-22
specifying fetch size for a crosstab view, example, 9-26
specifying fetch size for a table view, example, 9-25
specifying the behavior of, 8-8, 9-19
starting and ending positions of a value, example of calculating, 9-21
structure, 8-5
CursorInput class, 8-9, 8-11
CursorManager class, 8-12
hierarchy, 8-13
CursorManager objects
closing before rolling back a Transaction, 7-9
creating, example, 9-2
updating the CursorManagerSpecification, 8-12
CursorManagerSpecification class, 8-9
creating object, example, 9-2
CursorManagerUpdateEvent class, 8-15
CursorManagerUpdateListener class, 8-15
CursorSpecification class, 8-10
CursorSpecification objects
getting from a CursorManagerSpecification, example, 9-19

D

data store
definition, 1-4
exploring, 4-2
scope of, 4-2
data type
of MDM metadata objects, 2-24
of MdmSource objects, 2-26
data warehouse, 1-3
DataProvider objects
creating, 3-4
needed to create MdmMetadataProvider, 4-3
Date OLAP API data type, 2-25, 5-9, 5-10
default hierarchy
example of getting, 4-8
retrieving, 6-13
derived Source objects
definition, 5-5
description, 5-8
introduced, 5-8
dimensions
definition, 1-2
in OLAP metadata, 2-3
MdmDimension objects, 2-8
Source objects for, 5-6
distinct method
description, 5-7
documentation, A-3
Double OLAP API data type, 2-25, 5-9, 5-10
drilling down a hierarchy, 6-15
DriverManager objects, 3-3
dynamic queries, 10-2
DynamicDefinition class, 10-7

E

elements
of a level MdmHierarchy, 2-13
of a union MdmHierarchy, 2-16
of an MdmAttribute, 2-23
of an MdmLevel, 2-11
of an MdmListDimension, 2-18
of an MdmMeasure, 2-20
ranking, 6-10 to 6-12
selecting by value, 6-2
sorting, 6-10 to 6-12
Empty OLAP API data type, 2-26, 5-9
ExpressTransactionProvider class, 7-8

extent of a Cursor
definition, 8-25
example of calculating, 9-21
use of, 8-26
extract method, description, 5-7

F

faster varying Cursor components, 8-6
fetch block of a Cursor
definition, 8-27
determining shape of, 8-29
sharing, 8-29
fetch size of a Cursor
definition, 8-27
example of specifying, 9-25, 9-26
reasons for specifying, 8-27
specifying, 8-27
Float OLAP API data type, 2-25, 5-9, 5-10
fundamental Source objects
creating, 5-10
definition, 5-5
FundamentalMetadataObject class, 2-24
FundamentalMetadataProvider class, 2-24

G

getSource method
for creating primary Source objects, 5-5 to 5-7
for getting Source produced by a Template, example, 10-14
in DynamicDefinition class, 10-2, 10-7
in MdmSource class, 2-7
simple example, 4-9

H

hierarchies
creating Source objects for, 6-13
definition, 1-2
drilling down, 6-15
in OLAP metadata, 2-3
MdmHierarchy objects, 2-12
node and leaf terminology, 2-16
retrieving default, 4-8, 6-13

I

input-output order
determining, 6-4, 6-5
effect on Source structure, 6-4, 6-5
inputs
changing to outputs, 6-3 to 6-5
installation for application development, A-2
Integer OLAP API data type, 2-25, 5-9, 5-10

J

Java Development Kit, version required, A-2
Javadoc, A-3
JDBC
Connection objects, 3-3
DriverManager objects, 3-3
installing, A-2
loading drivers, 3-3
join method
changing inputs to outputs, 6-3
example, 6-4 to 6-5, 6-6, 6-16

L

leaf in a hierarchy, 2-16
level MdmHierarchy, 2-12
levels
definition, 1-2
in OLAP metadata, 2-3
MdmLevel objects, 2-10
list Source objects, 5-5

M

MDM. *See* multidimensional metadata model
MdmAttribute objects
creating Source objects for, 5-7
description, 2-23
elements, 2-23
example of getting, 4-8

MdmDimension objects
description, 2-8
example of getting related objects, 4-7
introduction, 1-6
regions, 2-9
related MdmAttribute objects, 2-9
related MdmDimensionDefinition objects, 2-9
related MdmDimensionMemberType objects, 2-10
MdmDimensionDefinition objects
description, 2-9
example of getting, 4-8
MdmDimensionMemberType objects
description, 2-10
example of getting, 4-8
MdmHierarchy objects
creating Source objects for, 6-13
description, 2-12
elements of a level MdmHierarchy, 2-13
elements of a union MdmHierarchy, 2-16
level type description, 2-12
union type description, 2-12
value type description, 2-12
MdmLevel objects
description, 2-10
elements, 2-11
MdmListDimension objects
description, 2-18
elements, 2-18
MdmMeasure objects
description, 2-19
elements, 2-20
example of getting their dimensions, 4-7
introduction, 1-6
kinds of values, 2-20
MdmMetadataProvider objects
creating, 4-4
description, 4-3
introduction, 1-6
MdmObject class, 2-5
MdmSchema objects
description, 2-6
getting contents of, 4-6
getting the root, 4-6
introduction, 1-6
root, 2-6, 4-4
MdmSource objects, 2-7
measure folders
in OLAP metadata, 2-4
mapped to MdmSchema objects, 2-6
measure MdmDimension objects, 4-6
measures
definition, 1-2
in OLAP metadata, 2-3
MdmMeasure objects, 2-19
Source objects for, 5-7
metadata
definition, 1-3
discovering, 4-2
distinguished from data, 1-5
mapping OLAP metadata to MDM metadata, 2-6
preparation for OLAP API, 1-3, 2-2
sample code for discovering, 4-9 to 4-26
MetadataState class, 10-6
example of implementation, 10-12
multidimensional metadata model (MDM)
description, 2-2
introduction, 1-5

N

nested outputs
getting values from a Cursor with, example, 9-8
of a Source, definition, 9-3
node in a hierarchy, 2-16
Null OLAP API data type, 5-9
Number OLAP API data type, 2-26, 5-9, 5-10
numeric comparisons
performing, 6-22
numeric functions
creating, 6-27
numeric methods
using, 6-23 to 6-29

numeric operations
  example, 6-20, 6-21
  list of methods for, 6-19, 6-22
  performing, 6-19 to 6-21

O

OLAP API
  definition, 1-2
  installing for application development, A-2
  software components, 1-7
OLAP API data types
  for MDM metadata objects, 2-24
  objects that represent, 5-9, 5-10
OLAP Metadata API, 2-2
OLAP metadata objects, 2-2
outputs
  changing from inputs, 6-3 to 6-5
  getting from a CompoundCursor, example, 9-5
  getting from a CompoundCursorSpecification, example, 9-19
  getting nested, example, 9-8
  in a CompoundCursor, 8-5, 8-23, 8-25
    positions of, 8-17

P

parameterized selections
  creating, 6-27
parameters
  creating, 6-27
parent attributes
  example of getting, 4-8
  for MdmHierarchy objects, 2-12
  for MdmLevel objects, 2-10
parent-child relationships
  creating Source objects for, 6-14
  in hierarchies, 2-3, 2-8, 2-10, 2-12, 2-16
position
  parent starting and ending, 8-22
position method
  description, 5-7
  example, 6-8
positions
  CompoundCursor, 8-17
  Cursor, 8-16
  of elements, 6-9
  ValueCursor, 8-16
primary Source objects
  definition, 5-5
  for parent-child relationship, 6-14
  from MdmHierarchy objects, 6-13
  from MdmSource objects, 2-7
  getting, 5-5 to 5-7
  result of getSource method, 4-9
  structure, 5-6, 5-7
primitive methods, 5-7, 5-8

Q

queries
  dynamic, 10-2
  Source objects that are not, 8-3
  steps in retrieving results of, 9-2

R

range Source objects, 5-5
ranking elements, 6-10 to 6-12
read Transaction object, 7-3
region attributes
  example of getting, 4-8
  for MdmHierarchy objects, 2-12
regions
  example of getting, 4-8
  of an MdmDimension, 2-9
relationships
  Source objects for, 6-14
root MdmSchema
  description, 2-6
  function of, 4-4
  obtaining, 4-6

S

Sales History schema
  accessing through sample program, A-3
  list of metadata objects in, 4-16
  metadata discovery program, 4-9
sample program, A-3

selecting elements
based on element values, 6-2 to 6-6
based on hierarchical position, 6-12 to 6-15
based on rank, 6-6 to 6-12
selectValue method
example, 6-6
self-relation
Source object for, 6-16
Short OLAP API data type, 2-25, 5-9, 5-10
slower varying Cursor components, 8-6, 8-20
software provided, A-2
sorting elements, 6-10 to 6-12
Source class
convenience methods, 5-8
primitive methods, 5-7, 5-8
shortcut methods, 5-8
Source methods
alias, 5-7
distinct, 5-7
extract, 5-7
position, 5-7
string, 6-29 to 6-31
value, 5-7
Source objects
active in a Transaction object, 7-2, 8-4
constant, 5-5
derived, 5-5
for attributes, 5-7
for measures, 5-7
for relationships, 6-14
for self-relation, 6-16
fundamental, 5-5
getting, 5-5 to 5-7
getting a modifiable Source from a DynamicDefinition, 10-7
list, 5-5
modifiable, 10-2
primary, 5-5
range, 5-5
structure, 5-6, 5-7
SourceGenerator class, 10-6
example of implementation, 10-13
span of a value in a Cursor
definition, 8-22, 9-21
SpecifiedCursorManager objects
closing, 8-12
returned by the createCursorManager method, 8-12
string methods, 6-29 to 6-31
String OLAP API data type, 2-25, 5-9, 5-10
subschemas
description, 4-4
getting contents, 4-6

T

table view
navigating Cursor for, example, 9-10
Template class, 10-6
designing, 10-7
example of implementation, 10-9
Template objects
benefits of using, 1-10
classes used to create, 10-3
for creating modifiable Source objects, 10-2
introductory example, 1-11
relationship of classes producing a dynamic Source, 10-4
Transaction objects used in, 7-5
Transaction objects
child read and write, 7-3
committing, 7-3
creating a Cursor in the current, 8-4
current, 7-2
example of using child, 7-9
getting the current, 7-8
preparing, 7-3
read, 7-3
rolling back, 7-7
setting the current, 7-8
using in Template classes, 7-5
write, 7-3
TransactionProvider interface, 7-8
TransactionProvider objects
creating, 3-4
needed to create MdmMetadataProvider, 4-3
tuple
definition, 8-17
in a Cursor, example, 9-9

type of an MDM object
defined, 2-28
obtaining, 2-29

U

union MdmHierarchy, 2-12

V

value MdmHierarchy, 2-12
value method, 5-7
Value OLAP API data type, 2-26, 5-9, 5-10
ValueCursor objects
getting from a parent CompoundCursor, example, 9-5
getting values from, example, 9-4, 9-5
position, 8-16
virtual Cursor
definition, 8-27
Void OLAP API data type, 2-26

W

write Transaction object, 7-3