Oracle9*i* OLAP

Developer's Guide to the OLAP DML

Release 2 (9.2)

March 2002

Part No. A95298-01

ORACLE®

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

Part No. A95298-01

Copyright © 2001, 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*, PL/SQL, and SQL*Plus are trademarks or registered trademarks of Oracle Corporation. Other names may be trademarks of their respective owners.

Contents

Send Us Your Comments xv

Preface xvii

Audience xviii
Organization xviii
Related Documentation xix
Conventions xx
Documentation Accessibility xxiii

What's New in the OLAP DML? xxv

Oracle9*i* Release 2 (9.2) New Features in the OLAP DML xxvi

Part I Introduction

1 Basic Concepts

What Is the OLAP DML? 1-2
Analytic Workspaces 1-2
SQL and the OLAP DML 1-3
The OLAP API and the OLAP DML 1-3
Using the OLAP DML 1-3
How to Use the OLAP DML to Analyze Data 1-4
Creating an Analytic Workspace 1-4
Loading Data Into Analytic Workspaces 1-5

Temporary vs. Persistent Analytic Workspaces 1-5
Sharing Data In Analytic Workspaces 1-5
Accessing a Workspace from OLAP Worksheet 1-6
Procedures: How to Open OLAP Worksheet 1-6
Establishing a Connection 1-7
Executing Commands 1-8
Editing an OLAP DML Program 1-8
Closing the Connection 1-9
Accessing a Workspace from SQL-Based Applications 1-9
Using SQL SELECT Statements 1-9
Using Embedded OLAP DML Commands 1-10
Accessing a Workspace from a Java Application 1-10
Using OLAP Metadata 1-10
Using Embedded OLAP DML Commands 1-10

2 Defining and Working with Analytic Workspaces

Using the OLAP DML to Work with Analytic Workspaces 2-2
Current Analytic Workspace 2-2
How to Create An Analytic Workspace 2-3
How to Attach an Analytic Workspace 2-3
Specifying the Analytic Workspace Attachment Mode 2-4
Sharing Analytic Workspaces 2-4
How to Detach an Analytic Workspace 2-5
How to Delete an Analytic Workspace 2-5
Workspace Localization Settings 2-6
Attaching Multiple Analytic Workspaces 2-6
Qualified Object Names 2-6
Multiple AUTOGO and Permission Programs 2-7
Using Names and Aliases for Analytic Workspaces 2-7
Workspace Names 2-7
Workspace Aliases 2-8
Saving Analytic Workspace Changes 2-8
UPDATE Command 2-9
COMMIT Command 2-9

Effect of the ROLLBACK Command 2-10
Minimizing Analytic Workspace Growth 2-10
Executing Programs Automatically 2-11
Program Names 2-11
AUTOGO Program Example 2-11
Adding Security to an Analytic Workspace 2-12
Permission Programs 2-12
Creating and Designing Permission Programs 2-13
Importing and Exporting Workspace Objects 2-14
Obtaining Analytic Workspace Information 2-15
Obtaining General Information About an Analytic Workspace 2-15
Viewing Objects in an Analytic Workspace 2-16
Obtaining Information About Objects 2-17

3 Defining Data Objects

Overview: Defining Workspace Objects 3-2
Workspace Objects That You Can Define 3-3
Data Types 3-4
Numeric Data Types 3-4
Examples of Literal Numeric Values 3-5
Text Data Types 3-5
Escape Sequences 3-6
Examples of Literal Text Values 3-7
Boolean Data Type 3-7
Date Data Types 3-7
Defining Dimensions 3-8
Determining What Dimensions to Define 3-9
How Data For Simple Flat Dimensions Is Stored 3-10
Defining Dimension Surrogates 3-11
Differences Between Dimensions and Dimension Surrogates 3-12
Defining Relations 3-13
How Relations Are Dimensioned 3-13
How Relation Data Is Stored 3-14
Example: Relation Between Two Dimensions 3-15
Example: Self-relation 3-15

Defining Variables 3-16
Types of Variables 3-17
How Variable Data Is Stored 3-17
Defining Variables That Handle Sparse Data Efficiently 3-18
Definition: Composite 3-18
Why You Should Use Named Composites 3-19
How to Use Composites 3-19
Naming, Renaming, and Unnaming Composites 3-20
Adding Data to a Variable That Uses a Composite 3-20
Defining a Variable with a Single-Dimension Composite 3-22
Defining Hierarchical Dimensions and Variables That Use Them 3-22
Defining a Variable with a Hierarchical Dimension 3-23
Example: Variable with a Hierarchical Dimension 3-24
Defining Concat Dimensions and Variables That Use Them 3-25
Example: Variable with a Concat Dimension 3-26
Changing the Definition of an Object 3-27

4 Working with Expressions

Introducing Expressions 4-2
Data Types of Expressions 4-2
How the Data Type of an Expression is Determined 4-2
Changing the Data Type of an Expression 4-3
Operators 4-3
Saving an Expression 4-4
Dimensionality of Expressions 4-5
Determining the Dimensions of an Expression 4-5
How Dimension Status Affects the Results of Expressions 4-6
Specifying a Single Value for the Dimension of an Expression 4-6
Qualifying a Variable 4-7
Replacing a Dimension in a Variable 4-8
Qualifying a Relation 4-9
Qualifying a Dimension 4-10
Using Ampersand Substitution with QDRs 4-10
Using the QUAL Function to Specify a QDR 4-10

Using Workspace Objects in Expressions 4-12
Using Dimensions or Dimension Surrogates in Expressions 4-12
Using Composites in Expressions 4-13
Using Variables in Expressions 4-13
Using Variables Defined with Composites in Expressions 4-14
Default Behavior of Commands That Loop Over Variables 4-14
Using Relations In Expressions 4-15
Using Functions in Expressions 4-15
Numeric Expressions 4-15
Arithmetic Operators 4-16
Mixing Numeric Data Types 4-17
Automatic Conversion of Numeric Data Types 4-17
Using Dimensions in Arithmetic Expressions 4-18
Using Dates in Arithmetic Expressions 4-18
Limitations of Floating Point Calculations 4-18
Controlling Errors During Calculations 4-19
Text Expressions 4-20
Working with Dates in Text Expressions 4-20
Working with NTEXT Data 4-21
Boolean Expressions 4-21
Creating Boolean Expressions 4-22
Comparing NA Values in Boolean Expressions 4-24
Controlling Errors When Comparing Numeric Data 4-24
Controlling Errors Due to Numerical Precision 4-25
Controlling Errors When Comparing Floating Point Numbers 4-25
Controlling Errors When Comparing Different Numeric Data Types 4-25
Comparing Dimension Values 4-26
Comparing Dates 4-27
Comparing Text Data 4-27
Comparing a Text Value to a Text Pattern 4-28
Comparing Text Literals to Relations 4-29
Conditional Expressions 4-29
Substitution Expressions 4-30
Working with NA Values 4-32
Controlling how NA values are treated 4-32

Working with the NATRIGGER Property 4-33
Using NASKIP 4-33
Using NASKIP2 4-34
Using NAFILL 4-34

5 Populating Workspace Data Objects

Overview: Populating an Analytic Workspace 5-2
Maintaining Dimensions and Composites 5-3
How Maintaining a Dimension Affects Dimension Status 5-4
Avoiding Deferred Maintenance 5-4
Adding Values to Dimensions 5-4
Updating Relations When Merging New Values 5-6
Deleting Values from Dimensions 5-7
Deleting Values from Conjoint Dimensions 5-8
Changing the Position of Dimension Values 5-8
Storing Dimension Values in Sorted Order 5-8
Maintaining Composites and Conjoint Dimensions 5-9
Maintaining Composites 5-10
Maintaining Conjoint Dimensions 5-10
Maintaining Concat Dimensions 5-10
Assigning Values to Data Objects 5-10
Using Objects in Assignment Statements 5-11
How Values Are Assigned to Variables with Composites 5-12
Assigning Values to Relations 5-14
Assigning Values to Dimensions 5-14
Assigning Values to Specific Cells of a Data Object 5-14
Calculating and Analyzing Data 5-15

6 Selecting Data

Introducing Dimension Status 6-2
Changing the Current Status List 6-2
Changing the Default Status List 6-2
Identifying and Retrieving Status Lists 6-3
Saving and Restoring Dimension Status 6-4
Limiting to a Simple List of Values 6-4

Limiting Using a Boolean Expression 6-5
How LIMIT Handles Boolean Multidimensional Expressions 6-6
Limiting to Values That Match an Expression 6-8
Limiting to the Top or Bottom Values 6-9
Limiting to the Values of a Related Dimension 6-11
How Limiting to a Related Dimension Determines Status 6-12
Suppressing the Sort When Limiting to a Related Dimension 6-12
Limiting Based on the Position of a Value in a Dimension 6-12
Limiting Using Value Position in its Dimension 6-12
Limiting Using Value Position in an Unrelated Dimension 6-13
Limiting Based on a Relationship Within a Hierarchy 6-13
Differences Between HIERARCHY and DESCENDANTS Keywords 6-14
Limiting Composites and Conjoint Dimensions 6-18
Ways of Limiting Conjoint Dimensions 6-19
Limiting Conjoint Dimensions Using Value Combinations 6-19
Limiting Conjoint Dimensions Using Base Dimension Values 6-20
Limiting Concat Dimensions 6-20
Working with Null Status 6-21
Managing Null Status in a Program 6-21
Errors When Limiting Status to a Null Value 6-21
Working with Valuesets 6-22
Creating a Valueset 6-22
Limiting Using a Valueset 6-23
Changing the Values of a Valueset 6-24
Identifying and Retrieving the Values in a Valueset 6-25
Retrieving the Values in a Valueset 6-25
Retrieving the Dimension Positions of Values in a Valueset 6-25

Part II Applications Development

7 Developing Programs

Introduction to OLAP DML Programs 7-2
Executing Programs 7-2
Executing User-Defined Functions 7-3

Defining and Editing Programs 7-3
Formatting Guidelines for Editing Programs 7-4
Using Variables in Programs 7-4
Global Versus Modular Design Approaches 7-5
Defining Temporary Variables 7-5
Defining Local Variables 7-6
Passing Arguments 7-7
Using the ARGUMENT Command 7-7
Using Multiple Arguments 7-8
Passing Arguments as Text with Ampersand Substitution 7-9
Passing Object Names and Keywords 7-11
Writing User-Defined Functions 7-11
Data Type of a User-Defined Function 7-12
Arguments in a User-Defined Function 7-12
Controlling the Flow of Execution 7-14
Guidelines for Constructing a Label 7-14
Alternatives to the GOTO Command 7-15
Directing Output 7-17
Capturing Error Messages 7-19
Preserving the Session Environment 7-19
Changing the Program Environment 7-19
Ways to Save and Restore Environments 7-20
Saving the Status of a Dimension or the Value of an Option 7-20
Saving Several Values at Once 7-21
Using Level Markers 7-21
Using CONTEXT to Save Several Values at Once 7-22
Handling Errors 7-23
How An Error Is Signaled 7-23
How An Error Is Trapped 7-23
Handling Errors While Saving the Session Environment 7-23
Suppressing Error Messages 7-24
Identifying the Error That Occurred 7-24
Creating Your Own Error Messages 7-25
Handling Errors in Nested Programs 7-26

Compiling Programs ... 7-28
Finding Out If a Program Has Been Compiled ... 7-29
Programming Methods That Prevent Compilation ... 7-29
Testing and Debugging Programs ... 7-29
Generating Diagnostic Messages ... 7-30
Identifying Bad Lines of Code ... 7-30
Sending Output to a Debugging File ... 7-31
Creating a debugging file ... 7-31
Specifying the contents of the debugging file ... 7-31

8 Working with Models

Using Models to Calculate Data ... 8-2
How Dimension Values Are Treated in a Model ... 8-3
Creating a Nested Hierarchy of Models ... 8-4
Working with the INCLUDE Command ... 8-5
Basic Modeling Commands ... 8-5
Writing Equations in a Model ... 8-6
Writing DIMENSION and INCLUDE Commands ... 8-6
Compiling a Model ... 8-7
Simple Blocks ... 8-8
Step Blocks ... 8-8
Simultaneous Blocks ... 8-9
Running a Model ... 8-9
Using Data from Past and Future Time Periods ... 8-10
Solving Simultaneous Equations ... 8-10
Debugging a Model ... 8-11
Modeling for Multiple Scenarios ... 8-12
Building a Scenario Model ... 8-12

9 Allocating Data

Introduction to Allocation ... 9-2
Preparing for an Allocation ... 9-5
Creating an Aggregation Map for Allocation ... 9-5
Using the Allocation Operators and Arguments ... 9-7
Using the HEVEN and MAX Operators and the ADD Argument ... 9-8

Using the COPY Operator and the PROTECT Argument 9-10
Using the HFIRST and HLAST Operators 9-13
Using the PROPORTIONAL Operator 9-15

Part III Analytic Workspace Management

10 Working with Relational Tables

Issuing SQL Statements Through the OLAP DML 10-2
Supported SQL Statements 10-2
Unsupported SQL Statements 10-2
Creating an Analytic Workspace from Relational Tables 10-3
Process: Designing and Defining an Analytic Workspace to Hold Relational Data 10-3
Process: Writing Programs that Populate Analytic Workspaces with Relational Data 10-4
Declaring a Cursor 10-5
Example: Declaring a Cursor 10-6
Using Variables in the WHERE Clause of the SELECT Statement 10-6
Using Conjunctions in a WHERE Clause 10-7
Opening a Cursor 10-8
Importing and Fetching Relational Table Data into Analytic Workspace Objects 10-8
Example: Copying Relational Table Data into Analytic Workspace Objects 10-11
Closing a Cursor 10-13
Cleaning up the SQL Cursors 10-14
Example: Creating an Analytic Workspace from Sales History Tables 10-14
Designing and Defining an Analytic Workspace for Sales History Data 10-15
Populating Analytic Workspace Objects with Sales History Data 10-19
Writing Data from Analytic Workspace Objects into Relational Tables 10-28
Using SQL PREPARE and SQL EXECUTE 10-29
Performing a Direct Insert 10-29
Inserting Workspace Data into Relational Tables: Example 10-29
Conditionally Updating a Relational Table 10-31
Using Stored Procedures and Triggers 10-32
Executing a stored procedure 10-33
Checking for Errors 10-34
SQLCODE Option 10-34
SQLERRM Option 10-34

SQLMESSAGES Option 10-35

11 Reading Data from Files

Introducing Data-Reading Programs 11-2
Reading Files 11-3
Creating a Program to Read Data 11-4
Specifying File Names in the OLAP DML 11-4
Reading Data from Files 11-5
Reading Structured PRN Files 11-6
Reading and Maintaining Dimension Values 11-7
Adding New Dimension Values from a Data File 11-9
Reading Dimension Values by Position 11-10
The Use of Composites 11-10
Reading and Maintaining Conjoint Dimensions 11-10
Translating Coded Dimension Values 11-11
Processing Input Data 11-14
Specifying a Conversion Type for Data 11-15
Processing Records Individually 11-15
Reading Different Records 11-17
Processing Several Values for One Variable 11-17

12 Aggregating Data

About Aggregating Detail Data 12-2
Functionality Available with AGGREGATE 12-2
Process Overview: Aggregation 12-4
Preliminary Steps Prior to Aggregation 12-4
Identifying the Parent and Level Relations 12-4
Verifying That All Composites Use BTREE Indexes 12-6
Creating an Aggregation Map 12-6
How to Define an Aggmap Object 12-7
How to Add Contents to an Aggmap Object 12-7
Contents of an Aggregation Map 12-9
How to Compile an Aggregation Map 12-10
Aggregating Multiple Variables with a Single Command 12-11

About the RELATION Command 12-12
Specifying an Aggregation Method 12-14
Selecting Data For Aggregation 12-16
Caching Runtime Aggregates 12-17
Aggregating Non-Hierarchical Data 12-18
How to Generate Precalculated Data 12-20
Effects of Dimension Status 12-21
Monitoring Progress 12-21
How to Calculate Data at Runtime 12-22
Setting Up Calculation on the Fly 12-22
Adding the $NATRIGGER Property to a Variable 12-23
Creating Custom Aggregates 12-23
Balancing Precalculated and Runtime Aggregation 12-24
Selecting Dimensions for Runtime Calculation 12-26
Selecting Levels for Runtime Calculation 12-27
Performing Partial Aggregations 12-27
Aggregation Changes That Cause Problems 12-28
Incremental Data Loading 12-28
Problem: PRECOMPUTE Status List Is Inaccurate 12-29
Solution: Regenerate the PRECOMPUTE Status List 12-29
Using a Data-Dependent PRECOMPUTE Clause 12-29
Problem: Values of the Limit Clause Vary With Each Data Update 12-30
Solution: Maintain a Valueset 12-30
Changing a Hierarchy 12-32
Problem: Previously Aggregated Data is Incorrect 12-33
Solution: Re-Aggregate Changed Branches 12-33
How to Aggregate Branches of a Hierarchy 12-34
Combining AGGREGATE with Forecasts and Programs 12-34
When to Use Multiple Aggregation Maps 12-35
Problem: Different Aggregation Maps Generate Different Status Lists 12-35
Solution: Create a Separate AGGMAP for the AGGREGATE Function 12-36

Index

Send Us Your Comments

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

Part No. A95298-01

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 Graaff 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 DML* provides an overview of the programming environment, describes workspace objects, and explains how to use the features of the OLAP DML. It also describes how to write and debug programs and illustrates programming strategies for accessing and working with data.

Note: All of the OLAP DML commands discussed in this guide are explained fully in the Oracle9i OLAP DML Reference help. For detailed information about a specific command, search for it by name in the reference.

This preface contains these topics:

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

Audience

Oracle9i OLAP Developer's Guide to the OLAP DML is intended for users who perform the following tasks:

- Access multidimensional data
- Perform analysis using the OLAP DML
- Manage analytic workspaces

To use this document, previous programming experience is helpful but not necessary.

Organization

This document contains:

Part I, Introduction

Chapter 1, "Basic Concepts"
Introduces the OLAP data manipulation language and describes various methods of accessing it.

Chapter 2, "Defining and Working with Analytic Workspaces"
Explains how to create new analytic workspaces and modify existing ones. Also describes initialization programs and password protection.

Chapter 3, "Defining Data Objects"
Describes the various types of workspace objects and how to create them. Defines workspace data types.

Chapter 4, "Working with Expressions"
Explains how to define and use expressions.

Chapter 5, "Populating Workspace Data Objects"
Explains how to add, delete, and reorder dimension members and assign values to data objects.

Chapter 6, "Selecting Data"
Explains how to select data for analysis or display.

Part II, Applications Development

Chapter 7, "Developing Programs"

Explains how to create, modify, compile, and run DML stored procedures.

Chapter 8, "Working with Models"

Explains how to create, compile, and run a series of equations.

Chapter 9, "Allocating Data"

Explains how to distribute data from parents to children in one or more dimensions.

Part III, Analytic Workspace Management

Chapter 10, "Working with Relational Tables"

Explains how to fetch data from relational tables into workspace objects, and how to insert data from workspace objects into relational tables.

Chapter 11, "Reading Data from Files"

Explains how to copy data from flat files into workspace objects.

Chapter 12, "Aggregating Data"

Explains how to roll up low-level data.

Related Documentation

For more information, see these Oracle resources:

- Oracle9i OLAP DML Reference help
- *Oracle9i OLAP User's Guide*
- *Oracle9i OLAP Developer's Guide to the OLAP API*
- Oracle9i OLAP API Javadoc
- *Oracle9i Data Warehousing Guide*

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 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**.

Convention	Meaning	Example
Italics	Italic typeface indicates book titles or emphasis.	*Oracle9i Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`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.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.
`lowercase monospace (fixed-width) font`	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the datafiles and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true.` Connect as `oe` user. The `JRepUtil` class implements these methods.
`lowercase italic monospace (fixed-width) font`	Lowercase italic monospace font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run *`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate 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:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

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

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`
...	Horizontal ellipsis points indicate either:	
	■ That we have omitted parts of the code that are not directly related to the example	`CREATE TABLE ... AS `*`subquery;`*
	■ That you can repeat a portion of the code	`SELECT `*`col1, col2, ... , coln`*` FROM employees;`

.
.
.
	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	`SQL> SELECT NAME FROM V$DATAFILE;`
		`NAME`
		`------------------------------------`
		`/fsl/dbs/tbs_01.dbf`
		`/fs1/dbs/tbs_02.dbf`
		`.`
		`.`
		`.`
		`/fsl/dbs/tbs_09.dbf`
		`9 rows selected`

Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);` `acct    CONSTANT NUMBER(4) := 3;`
Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/`*`system_password`* `DB_NAME = `*`database_name`*

UPPERCASE	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

Convention	Meaning	Example

`lowercase`	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		
		`sqlplus hr/hr`
		
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

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.

What's New in the OLAP DML?

Oracle9*i* Release 2 provides the OLAP data manipulation language (DML) for multidimensional analysis within the Oracle database. With the OLAP option installed, you can execute DML commands for manipulating data in an analytic workspace. Users of Oracle Express Server release 6.3 will find that there are some new and changed features in the OLAP DML.

See Also:

- *Oracle9i OLAP User's Guide* for general information about the OLAP option in Release 2 and for specific differences between Express Server and Oracle OLAP.
- Oracle9i OLAP DML Reference help for lists of added, deleted, renamed, and significantly changed commands in the OLAP DML.

The following section describes the new features in Oracle9*i* OLAP:

- Oracle9i Release 2 (9.2) New Features in the OLAP DML

Oracle9*i* Release 2 (9.2) New Features in the OLAP DML

The following list briefly describes the new features of the OLAP DML.

- **The DATABASE command and related commands have been renamed.**

  Because the OLAP engine runs in the Oracle kernel and analytic workspaces are stored in relational tables, there is no separate file that stores an analytic workspace. This change is reflected in new names and new functionality for existing DML commands.

  > **See Also:** Chapter 2, "Defining and Working with Analytic Workspaces"

- **Access to the OLAP DML is through SQL and the OLAP API, not through XCA, SNAPI, or ODBC.**

  XCA, SNAPI, and ODBC connections are no longer supported, and the related commands have been removed. Note that session sharing is not supported in the new access methods.

  > **See Also:** The *Oracle9i OLAP User's Guide* for information about SQL access and the *Oracle9i OLAP Developer's Guide to the OLAP API*

- **In order to save changes to an analytic workspace, you must use both the UPDATE and COMMIT commands.**

  The `UPDATE` command moves changes from a temporary area to the dataase table in which the workspace is stored. The changes are not saved until you execute a `COMMIT` command, either from the OLAP DML or from SQL.

  > **See Also:** Chapter 2, "Defining and Working with Analytic Workspaces"

- **Custom aggregates are supported.**

  Virtual dimension members can be defined at runtime using the new `AGGREGATION` command within a `MODEL` object. The `AGGREGATE` function then calculates data for the custom aggregate the same as any other aggregate.

  > **See Also:** Chapter 12, "Aggregating Data"

- **Models can be used to aggregate data over nonhierarchical dimensions.**

  A MODEL command in an aggregation map executes a model either as a data maintenance step (using the AGGREGATE command) or at runtime (using the AGGREGATE function).

  > **See Also:** Chapter 12, "Aggregating Data"

- **Allocation of data over a hierarchy is supported.**

  A new ALLOCATE command provides support for planning applications, such as enterprise budgeting and demand planning systems, which need to allocate data to lower levels of a hierarchy based on sophisticated allocation rules.

  > **See Also:** Chapter 9, "Allocating Data"

- **SQL IMPORT command provides a high performance method of copying data from database tables into an analytic workspace.**

  The SQL IMPORT command loads fact data into workspace objects more quickly than an SQL FETCH statement.

  > **See Also:** Chapter 10, "Working with Relational Tables"

- **SQL PREPARE command provides high performance method of loading workspace data into database tables.**

  The SQL PREPARE command includes new options that you can use to specify direct-path insertion of analytic workspace data into relational tables.

  > **See Also:** Chapter 10, "Working with Relational Tables"

- **Concat dimensions join values from multiple dimensions into one dimension.**

  In defining a concat dimension, you can combine the values of two or more dimensions into one dimension. You can use a concat dimension to map multidimensional structures to relational schemas and thereby improve data loading from relational sources. You can also use concat dimensions in performing custom aggregations and other customized operations.

  > **See Also:** Chapter 3, "Defining Data Objects" and Chapter 10, "Working with Relational Tables"

- **NUMBER dimensions store numeric values other than ordinal integers.**

  The Oracle OLAP DML has a new `NUMBER` data type that is the equivalent of the `NUMBER` data type in the relational database. You can define a `NUMBER` dimension that has `NUMBER` values. Oracle OLAP always interprets the values of a `NUMBER` dimension as dimension values and not as ordinal position values. You can use a `NUMBER` dimension to represent a series of unique numeric values, such as a surrogate key column in a relational database table.

  See Also: Chapter 3, "Defining Data Objects"

- **Dimension surrogates provide alternative labels for dimension values.**

  A dimension surrogate is a new type of DML object. You define a dimension surrogate based on a dimension, but the surrogate can be of a different data type than its dimension. The surrogate has the same number of positions as the dimension. You assign values to a surrogate as you would to a variable. You can use a `NUMBER` dimension and a dimension surrogate to load surrogate key values from a relational database into an analytic workspace, and then use those key values to load data from the relational fact table or tables into multidimensional structures.

  See Also: Chapter 3, "Defining Data Objects"

- **Qualified object names allow you to reference identically named objects in more than one attached analytic workspace.**

  In OLAP DML commands, you can specify an object using its qualified object name, which includes not only the name of the object but also the name of the analytic workspace in which the object resides.

  See Also: Chapter 2, "Defining and Working with Analytic Workspaces"

- **Full names for analytic workspaces allow you to access workspaces that belong to another user.**

  In OLAP DML commands, you can specify an analytic workspace that is in another user's shema by using the full name of the workspace. The full name includes the schema name.

  See Also: Chapter 2, "Defining and Working with Analytic Workspaces"

- **Workspace aliases make possible short and generic names for analytic workspaces.**

  Workspace aliases allow you to reference an analytic workspace using a name that is easier to type than its full name. Aliases also let you write generic code that includes a reference to a workspace but does not hard-code its name.

  > **See Also:** Chapter 2, "Defining and Working with Analytic Workspaces"

- **Directory aliases provide the mechanism through which OLAP DML commands can access files on disk.**

  When you read from or write to a disk file using the OLAP DML, you do not directly specify the directory in which the file resides. Instead, you specify a directory alias that has been set up for your use by the Oracle database administrator.

  > **See Also:** Chapter 11, "Reading Data from Files" and the *Oracle9i OLAP User's Guide*

- **The new NTEXT data type can hold data from NCHAR and NVARCHAR2 columns in the database.**

  All `NTEXT` values are encoded in the UTF8 Unicode transformation format.

  > **See Also:** Chapter 3, "Defining Data Objects"

- **The default character set for Oracle OLAP is the database character set.**

  Oracle OLAP no longer has a configuration setting that specifies the default character set. The Oracle OLAP default is the same as the databse character set.

- **Oracle OLAP NLS parameter settings are coordinated with the settings for the database.**

  All Oracle OLAP NLS settings (such as `NLS_DATE_FORMAT` and `NLS_LANGUAGE`) reflect the session-wide NLS parameter settings. If you set the NLS options in Oracle OLAP, you change your session-wide NLS parameter settings.

  > **See Also:** The NLS options in the Oracle9i OLAP DML Reference help

- **The DECIMALCHARS, THOUSANDSCHARS, YESSPELL, and NOSPELL options are read-only.**

  The values of these options always mirror the current session-wide NLS parameter settings. You cannot change these settings by changing the values of the Oracle OLAP options.

- **The commands that gave access to operating system activities are no longer supported**

  To be compatible with Oracle database conventions, Oracle OLAP does not provide direct access to system-level information and commands. Therefore, the `SYSINFO` function has fewer keywords, and commands such as `CHDIR`, `CHDRIVE`, `MKDIR`, and `SHELL` have been removed. In addition, EXTCALL objects are no longer supported.

  See Also: The list of deleted commands in the Oracle9i OLAP DML Reference help

- **In-place variables are no longer supported**

  Because analytic workspaces are stored in database tables, in-place variable storage is no longer applicable.

- **Interactive debugging is not supported.**

  You cannot use the `TRACE` and `WATCH` commands for interactive debugging in OLAP Worksheet, but you can use `PRGTRACE`, `MODTRACE`, and `DBGOUTFILE` to record the progress of your programs and models.

  See Also: Chapter 7, "Developing Programs" and Chapter 8, "Working with Models"

- **Performance statistics are available through relational views, instead of OLAP DML commands**

  The `DGCART` command and function as well as the `CACHEHITS`, `CACHEMISSES`, and `CACHETRIES` options have been removed. However, you can use OLAP dynamic performance views to monitor performance.

  See Also: The *Oracle9i OLAP User's Guide*

- **Forecasting capabilities have been enhanced with the addition of multi-cycle periodicity.**

  The `FCSET` command allows for multi-cycle periodicity in the forcasts created with `FCOPEN`, `FCCLOSE`, and `FCEXEC`.

  > **See Also:** The `FCOPEN`, `FCCLOSE`, `FCEXEC`, `FCQUERY`, and `FCSET` commands in the Oracle9i OLAP DML Reference help

- **Stripping of programs is no longer supported.**

  The `STRIP` command has been removed. Use the `HIDE` command instead. In previous releases, programs were stripped of their definitions in an analytic workspace file before it was delivered as part of an application. Thus, only compiled code was delivered. Now, analytic workspaces are delivered as EIF files, which contain only definitions and cannot contain compiled code. In this new context, stripped programs would not be executable.

Part I

Introduction

Part I describes the basic features of the OLAP DML.

It contains the following chapters:

- Chapter 1, "Basic Concepts"
- Chapter 2, "Defining and Working with Analytic Workspaces"
- Chapter 3, "Defining Data Objects"
- Chapter 4, "Working with Expressions"
- Chapter 5, "Populating Workspace Data Objects"
- Chapter 6, "Selecting Data"

1 Basic Concepts

This chapter provides an overview of the basic concepts that you should understand before you begin programming in the OLAP DML. It includes the following topics:

- What Is the OLAP DML?
- Using the OLAP DML
- Accessing a Workspace from OLAP Worksheet
- Accessing a Workspace from SQL-Based Applications
- Accessing a Workspace from a Java Application

What Is the OLAP DML?

The OLAP DML is a data manipulation language. You can use DML commands and functions to perform complex analysis of data. You can also write programs that contain DML commands and functions.

The basic syntactic units of the OLAP DML are:

- Commands that initiate actions
- Functions that initiate actions and return a value
- Options to which you assign a value and that can influence the analytic workspace processing environment in various ways

OLAP DML commands, functions, and options are collectively referred to as commands. This guide introduces many of these commands. For the complete syntax for each command, usage notes, and examples, consult in the Oracle9i OLAP DML Reference help.

The purpose of the OLAP DML is to enable application developers to extend the analytical capabilities of querying languages such as SQL and the OLAP API.

To describe the purpose of the OLAP DML, it is important to discuss a few important concepts such as:

- Analytic workspaces
- The relationship of SQL to the OLAP DML
- The relationship of the OLAP API to the OLAP DML

Analytic Workspaces

An analytic workspace is a multidimensional data source. It may be temporary (that is, for the life of the session), or it may be persistent. When an analytic workspace is persisted, the data is stored as LOBs in relational tables.

The multidimensional model of the analytic workspace is designed to support rapid and advanced calculations. Analytic workspaces also provide an alternative to materialized views as a means of storing aggregate data.

An application can access data that resides in an analytic workspace in either of two ways. One way is through PL/SQL packages that are provided by Oracle for access to analytic workspace data. The other way is through the Oracle OLAP API, which is a Java application programming interface. Both the PL/SQL packages and the

OLAP API provide ways to explicitly execute OLAP DML commands and programs.

SQL and the OLAP DML

SQL table functions can take a set of rows as input and produce a set of rows as output that can be queried like a physical database table. Oracle provides PL/SQL packages that use table functions to create views of multidimensional data residing in an analytic workspace. SQL applications can access these views. Thus, the calculation engine and analytic workspace data are accessible to SQL, making analytic and predictive functions available to SQL-based applications. SQL applications can connect to the database using either the Oracle Call Interface (OCI) or Java Database Connectivity (JDBC).

In addition to using PL/SQL procedures for accessing analytic workspace data as SQL views, application programmers can use the Oracle OLAP packages to directly execute OLAP DML commands and return the results to their applications.

The OLAP API and the OLAP DML

Java programs using the OLAP API can access data stored either in SQL tables or in analytic workspaces. The OLAP API provides a wide variety of analytic functions that allow the application to derive calculated measures from the data.

In some cases, however, the OLAP API does not provide the means to calculate data needed by an application. Examples include forecasts, solving a model, some types of consolidations (aggregations), and allocations. In these cases, you can directly execute OLAP DML commands from within the OLAP API to calculate this data within an analytic workspace.

Using the OLAP DML

The following are some situations in which you might use the OLAP DML:

- When you need to calculate data that cannot be calculated as part of your data warehouse extraction, transformation, and load (ETL) process or by using the Java OLAP API.
- When your application needs to perform and persist various calculations, but you do not want to immediately commit this calculation in SQL tables.
- When you want to manipulate data that is stored in an analytic workspace.

The most common types of calculations that the OLAP DML is used for include:

- Forecasts
- Models (a group of calculations in which the results of one calculation are used as input to another calculation)
- Allocations (a "reverse aggregation" in which you distribute data to lower levels based on a particular distribution scheme)
- Some types of non-additive aggregations (consolidations), such as hierarchical weighted averages

In addition, the OLAP DML can be used when you want to perform calculations that are not easily accomplished in the ETL process or by using the OLAP API.

You can commit data to the analytic workspace without committing it to SQL tables. This is very useful for work in process. For example, you might have a forecasting application where you want to allow users to save personal forecasts and reuse them during a later session, but you do not want users to commit the forecast to the SQL tables.

How to Use the OLAP DML to Analyze Data

To use the OLAP DML, you:

1. Create an analytic workspace.
2. Define data objects within the analytic workspace.
3. Load data into these objects.
4. Define and execute OLAP DML commands and programs.

After you use the OLAP DML to analyze data, you can then:

- View data in an analytic workspace using the OLAP API or SQL.
- Write data to SQL tables.

Creating an Analytic Workspace

You can create an analytic workspace with a command such as the following:

```
AW CREATE salesforecast
```

This command creates a new and empty analytic workspace named `salesforecast`.

For more information about creating an analytic workspace, refer to Chapter 2, "Defining and Working with Analytic Workspaces".

Loading Data Into Analytic Workspaces

To use the OLAP DML, data must exist in the analytic workspace. Data can be loaded into an analytic workspace from SQL tables or from flat files. In most cases, tables within the database will be the data source. To load data into the analytic workspace, you use commands in the OLAP DML.

For more information about loading data into an analytic workspace, refer to Chapter 10, "Working with Relational Tables" and Chapter 11, "Reading Data from Files".

Temporary vs. Persistent Analytic Workspaces

Analytic workspaces can be either temporary or persistent, depending on your needs. If the analytic workspace is needed only to perform a specific calculation and the results of the calculation do not need to be persisted in the workspace, the workspace can be discarded at the end of the session. This might occur if, for example, your application needs to forecast a small amount of sales data. Since the forecast can be rerun at any time, there might not be any point in persisting the results.

Analytic workspaces can also be persisted across sessions. You might want to persist data in the analytic workspace if you have calculated a significant amount of data (for example, a large forecast or the results of solving a model), or if you have aggregated data using non-additive aggregation methods.

Sharing Data In Analytic Workspaces

Data in analytic workspaces may be shared by many different users. To share data in an analytic workspace, the workspace needs to be persisted during the period of time it is to be shared.

For example, if you want to allow a user to share the results of a forecast, you can allow the user to persist the analytic workspace. If another user attaches that workspace during their application session, they can be allowed to see the other user’s forecast.

Accessing a Workspace from OLAP Worksheet

OLAP Worksheet is an interactive interface to Oracle OLAP that you can use to perform the following tasks:

- Connect to an analytic workspace
- Execute most OLAP DML commands
- Create and populate data objects
- Create, modify, compile, and execute DML programs
- Execute SQL statements

OLAP Worksheet has a Command Input window and a program Edit window.

You can enter commands in the query (input) pane at the bottom of the command input window and see results in the response (output) pane at the top.

Once you have opened OLAP Worksheet, you can use it to establish a connection to Oracle OLAP, open a workspace, execute OLAP DML commands or write and debug programs, save any changes, close the workspace, and close the connection.

Note: The following procedures identify menu choices that you can use to do various tasks. You may prefer to use the icons on the left side of the window, which provide a short-cut to some of the tasks.

Procedures: How to Open OLAP Worksheet

You can open OLAP Worksheet either from Oracle Enterprise Manager or from the operating system command line.

To open OLAP Worksheet from Oracle Enterprise Manager, take these steps:

1. Open Oracle Enterprise Manager and open a connection to your database.
2. Expand the database folder.
3. Right click on OLAP to see a menu, then choose **OLAP Worksheet**.

Tip: If you are unable to start OLAP Worksheet, then check system variables HOMEDRIVE and HOMEPATH. They do not need to be defined, but if they are, then they must be set to valid values.

On Unix, to open OLAP Worksheet from the command line, take these steps:

1. Using a command-line interface, go to the `bin` subdirectory of the OLAP Worksheet installation directory.
2. Run the `runapp.sh` script.

On Windows, click the OLAP Worksheet icon that was created during installation.

See Also: OLAP Worksheet Help for information about using OLAP Worksheet.

Establishing a Connection

Take these steps to establish a connection to Oracle OLAP:

1. From the OLAP Worksheet menu bar, choose **Server.**
2. Choose **Connect.**

   You see the Login to Database box.
3. Enter valid database user credentials and connection information in the Login dialog box that appears.

   In the Service box, type the identification of the Oracle database, in the following format:

   `host:port:SID`

   For example, `mycomputer:1521:rel9i`.

Oracle controls your access to data on the basis of your database user ID. Your user ID must have access rights to the analytic workspaces and relational tables that you want to use in OLAP Worksheet, or you will get an error when you try to access them.

Executing Commands

You can execute OLAP DML commands and SQL statements in the Command Input window of OLAP Worksheet.

By selecting different options in the Options menu, you can specify whether you want to execute OLAP DML commands or SQL statements. You can also specify whether you want commands executed individually or saved in a buffer and executed together.

- To execute OLAP DML commands, select **SQL Off** from the Options menu. To execute SQL statements, select **SQL On**.
- To execute commands as soon as you press Enter, select **Execute on Enter** from the Options menu.

  Or, to save the commands in a buffer, clear **Execute on Enter**. Then, to execute all of the commands that you have entered in the query pane, choose **Execute** from the View menu.

Be sure to locate your cursor in the query pane before you start to type. If you want to break a long command into several lines, you can continue the command on the next line by typing a continuation character (-) at the end of the current line.

When the SQL option is ON, just type in the SQL statements and press Enter. Do not terminate SQL statements with a semicolon. If you do, you will get an error.

Editing an OLAP DML Program

You can open a DML program in an Edit window so that you can add or modify program content. You can have multiple Edit windows open simultaneously, but an object definition can appear in only one Edit window at a time.

In addition to using the Edit window to work on a program, you can use it to edit a model or an aggregation map.

To edit a program, follow these steps:

1. In the input pane of the Command Input window, type

   ```
   edit object_name
   ```

   Where *object_name* is the name of a DML program object that already exists. Use the `DEFINE` command to create a new program object. If you want to edit a model or an aggregation map, type `MODEL` or `AGGMAP` before the object name.
2. In the Edit window, you can add, modify, or delete program content.

3. To save your changes, choose **Save** from the File menu. Note that this choice executes and UPDATE command, which updates all the changes that have been made in the analytic workspace up to this point.
4. To close the Edit window, choose **Quit** from the File menu.

> **See Also:** Chapter 7, "Developing Programs" for more information about DML programs.

Closing the Connection

Use the following procedure to close a connection to Oracle OLAP:

1. In the OLAP Worksheet menu bar, choose **Server.**
2. Choose **Disconnect.**
3. When prompted to disconnect, choose **Yes.**

When you disconnect, OLAP Worksheet executes a COMMIT command before ending your session. If you have executed the UPDATE command or chosen **Save** from the File menu of an Edit window before disconnecting, then the changes that you made before the update are made permanent. Otherwise, they are discarded. Any changes that you have made after the update are discarded when you disconnect.

Accessing a Workspace from SQL-Based Applications

SQL programmers can query data in the analytic workspace using SQL SELECT statements that use OLAP table functions and by embedding OLAP DML commands in their SQL scripts. The *Oracle9i OLAP User's Guide* describes these activities.

Using SQL SELECT Statements

SQL programmers can query analytic workspace data using SQL SELECT statements.

- If the analytic workspace has already been defined to the relational schema using the CWM2_OLAP_AW_ACCESS PL/SQL package, views of the analytic workspace have been created. You can query the analytic workspace by using SQL SELECT statements against these SQL views. This method requires minimum knowledge of the underlying data in the analytic workspace.

- You can query the analytic workspace using the OLAP_TABLE function in SQL `SELECT` statements. This method requires intimate knowledge of the analytic workspace data. The OLAP_TABLE function is provided with Oracle OLAP.

Using Embedded OLAP DML Commands

Using the procedures and functions in the DBMS_AW package, SQL programmers can issue OLAP DML statements against analytic workspace data. They can move data from relational tables into an analytic workspace, perform advanced analysis of the data (for example, forecasting), and move data from the analytic workspace back into relational tables.

Accessing a Workspace from a Java Application

Typically, a Java application uses the OLAP API to access relational data. In addition, the Oracle OLAP API supports access to data that resides in an analytic workspace. The *Oracle9i OLAP Developer's Guide to the OLAP API* and the OLAP API Javadoc describe these activities.

Using OLAP Metadata

Through the OLAP API, a Java application can access workspace data that has been exposed in OLAP metadata. Because OLAP metadata is compatible with the OLAP API multidimensional metadata (MDM) model, a Java application can manipulate workspace data using OLAP API Java classes. For information on how a database administrator exposes workspace data in OLAP metadata, see the *Oracle9i OLAP User's Guide.*.

Using Embedded OLAP DML Commands

The OLAP API provides a way for a Java application to directly manipulate workspace data, without the need for any metadata and without the use of the OLAP API data manipulation classes. The Java application uses the `SPLExecutor` class in the OLAP API to open a workspace and send DML commands directly to Oracle OLAP for execution in the workspace.

2

Defining and Working with Analytic Workspaces

This chapter discusses creating, attaching, and managing analytic workspaces. It includes the following topics:

- Using the OLAP DML to Work with Analytic Workspaces
- Attaching Multiple Analytic Workspaces
- Using Names and Aliases for Analytic Workspaces
- Saving Analytic Workspace Changes
- Executing Programs Automatically
- Adding Security to an Analytic Workspace
- Importing and Exporting Workspace Objects
- Obtaining Analytic Workspace Information

Using the OLAP DML to Work with Analytic Workspaces

To make the data and the object definitions of an analytic workspace available to your session, the analytic workspace must be attached. Analytic workspaces that are currently attached are known as **active analytic workspaces**. Attaching analytic workspaces is described in "How to Attach an Analytic Workspace" on page 2-3.

You can view a list of the active analytic workspaces by using the AW command with the LIST keyword.

```
AW LIST
```

This command displays a list of the active analytic workspaces. The express analytic workspace, which is a system analytic workspace that contains objects used internally, always appears in the analytic workspace list.

Current Analytic Workspace

The current analytic workspace is the first analytic workspace in the list of the active analytic workspaces that you view with the AW command with the LIST keyword. By default, when you define new workspace objects, they reside in the current analytic workspace, unless you specify the name of another active analytic workspace. Additionally, programs such as LISTNAMES list only the objects in the current analytic workspace. However, even when an active analytic workspace is not current, you can still change and update its data, edit and run its programs, and modify its object definitions.

Your session does not have to have a current analytic workspace. If you start Oracle OLAP without specifying an analytic workspace name, then the express analytic workspace is first on the list. However, the express analytic workspace is not current; there is no current analytic workspace until you specify one with the AW command.

You can retrieve the name of the current analytic workspace by using the AW function with the NAME keyword.

Suppose that you have two analytic workspaces attached, one named marketing and another named personnel. The following commands use the AW function with the NAME keyword to retrieve the name of the current analytic workspace into a

variable named MYTEXT, and then display the value of MYTEXT. This value is shown after the commands.

```
mytext = AW(NAME)
SHOW mytext

PERSONNEL
```

How to Create An Analytic Workspace

The AW command is used to create a new analytic workspace. The following example creates an analytic workspace named finance.

```
AW CREATE finance
```

When you create an analytic workspace, Oracle OLAP automatically executes a COMMIT command.

You are the only user who has access to a workspace that you have just created. If you want others to use the workspace, you must give them access to the relational table in which the workspace is stored. The name of the table is AW$ followed by the workspace name that you specified in your AW CREATE command.

To give read access to another user, execute a command like the following one in SQL. In this example, the workspace name is demo and the user's name is scott.

```
GRANT SELECT ON aw$demo TO scott
```

To give write access to another user, execute a SQL command like the following one.

```
GRANT UPDATE ON aw$demo TO scott
```

As in any SQL GRANT command, you can specify a group or role instead of a user.

How to Attach an Analytic Workspace

You can use the AW command to attach and detach analytic workspaces during a session. In addition, as you work in your session, you can use the AW command to switch freely among active analytic workspaces.

You attach an analytic workspace by using the AW command with the ATTACH keyword. The analytic workspace that you specify is automatically attached and made to be the current analytic workspace. The following example attaches an existing analytic workspace named finance and makes it the current analytic

workspace. Previously attached workspaces move down the list of attached workspaces to make room for the new one at the top of the list.

```
AW ATTACH finance
```

When you attach an analytic workspace, the default access to it is read-only. If you want a different attachment mode, then you must explicitly specify it in the AW command as described in "Specifying the Analytic Workspace Attachment Mode" on page 2-4.

Note: You can create programs that are automatically executed when you attach an analytic workspace. For more information, see "Executing Programs Automatically" on page 2-11.

Specifying the Analytic Workspace Attachment Mode

You can specify whether you want the analytic workspace attached in read-only mode, read/write nonexclusive mode, or read/write exclusive mode by using the RO, RW, and RX keywords of the AW command.

An analytic workspace that is attached in read/write nonexclusive mode or read-only mode can be accessed simultaneously by several sessions. However, only one session can have the analytic workspace open with read/write access. If another user has already attached an analytic workspace in read/write mode, then you cannot attach the same analytic workspace in read/write mode until that other user detaches it.

An analytic workspace that is attached in read/write exclusive mode cannot be accessed by any other session. If other users have already attached an analytic workspace, then you cannot attach the same analytic workspace in read/write exclusive mode until all of the other users detach it.

Sharing Analytic Workspaces

An analytic workspace can be accessed simultaneously by several sessions, assuming that the session users have been granted access by the creator of the workspace. Many sessions can access a workspace, but only one session can have it open with read/write access at any given time.

When you attach an analytic workspace, your default access to it is read only. Oracle OLAP supports simultaneous access for one writer and many readers of an analytic workspace. Provided your user ID has the appropriate access rights, you can always get read-only access to an analytic workspace, no matter how many

other users are using it. If another user has read/write access and commits changes to the analytic workspace, then your view of the analytic workspace does not change; you must detach and reattach the analytic workspace to see the changes.

If you want read/write access, then you must explicitly specify it in the AW command. If the analytic workspace is attached in read/write mode by another session, the response to your request for access depends on the keywords used in AW command.

You can specify whether or not you want to wait until an analytic workspace is available for the type of access you are you are requesting by using the WAIT and NOWAIT keywords of the AW command.

- If you specify the NOWAIT keyword (the default) and if the analytic workspace is not available for the type of access you are requesting, then an error message is produced that indicates that the analytic workspace is unavailable.
- If you specify the WAIT keyword and the analytic workspace is not available for the type of access you are requesting, then Oracle OLAP places you on the wait list for the analytic workspace.

How to Detach an Analytic Workspace

To detach an analytic workspace, you use the AW command with the DETACH keyword. The following command detaches the finance analytic workspace.

```
AW DETACH finance
```

A detached analytic workspace remains in the database. However, it is no longer accessible in your session. To access it again, use the AW command with the ATTACH keyword.

How to Delete an Analytic Workspace

To delete an analytic workspace from the database, you use the AW command with the DELETE keyword. Before deleting, you must detach the analytic workspace. The following commands delete the finance analytic workspace.

```
AW DETACH finance
AW DELETE finance
```

A deleted analytic workspace is no longer in the database; you can never access it again. When you delete an analytic workspace from the database, Oracle OLAP automatically executes a COMMIT command.

Workspace Localization Settings

Oracle supports locales that vary in their character sets, date formats, currency symbols, and other language-specific characteristics. Oracle globalization support is based on the value of parameters that begin with "NLS." For information about NLS parameters, see the *Oracle9i SQL Reference* and the *Oracle9i Database Globalization Support Guide*.

Within a session you can dynamically modify the value of some NLS parameters by setting them using the OLAP DML options that begin with "NLS." For example, you can set the value of `NLS_LANG` or `NLS_TERRITORY` in the OLAP DML. When you set the value of an OLAP DML NLS option, the setting affects your entire database session. It is not limited to your work in an analytic workspace.

Alternatively, you can use the following SQL command to change an NLS parameter for your entire session, including Oracle OLAP.

```
ALTER SESSION SET parameter = value
```

For more information about the OLAP DML NLS options, see the Oracle9i OLAP DML Reference help.

Attaching Multiple Analytic Workspaces

You can attach more than one analytic workspace at a time. However, when working with multiple analytic workspaces, you must take care when you name objects. When you request an object by name, either with the `DESCRIBE` command or by referring to it in a command or program, all the active analytic workspaces are searched until the named object is found. When you intend to use several analytic workspaces together, do not give the same name to objects in different analytic workspaces, unless you are prepared to use qualified object names when you reference the objects.

Qualified Object Names

When you attach more than one workspace, and objects in more than one workspace have duplicate names, you must use qualified object names to indicate which objects you want to reference.

A qualified object name uniquely identifies an object by including the workspace name. By using a qualified object name, you can clearly indicate to Oracle OLAP which object (in which workspace) you want to access.

For example, if you have attached the NORTHEAST workspace, which has a variable called SALES, and you have attached the SOUTHEAST workspace, which also has a variable called SALES, you must specify these variables using the following qualified object names (QONs).

```
northeast!sales
southeast!sales
```

The first part of a QON is a workspace name, and the second part is the name of the object. An exclamation point (!) joins the two parts.

You can intermix the use of qualified and unqualified names. You only need to use the qualified name to identify a specific object in one workspace when an object that has the same name exists in another attached workspace. If you do not specify a QON for either duplicate, then Oracle OLAP might use one or the other; the results are undefined.

Multiple AUTOGO and Permission Programs

If you have AUTOGO or permission programs defined in analytic workspaces that are currently attached, then the one in the analytic workspace that you are attaching is executed. However, if you have analytic workspace permission programs in more than one currently attached analytic workspace, then you must use their qualified object names when you edit them or use them in any other way. This will ensure that you access the appropriate version.

See Also: "Executing Programs Automatically" on page 2-11 for information on AUTOGO programs and "Adding Security to an Analytic Workspace" on page 2-12 for information on permission programs.

Using Names and Aliases for Analytic Workspaces

The OLAP DML provides alternative ways to refer to an analytic workspace to allow your code to be unambiguous and flexible.

Workspace Names

A workspace name is assigned when a workspace is created with the AW CREATE command. For example, in the command `aw create demo`, the workspace name is DEMO.

By default, a workspace is created in the schema for your database user ID. For example, if the user SCOTT created the DEMO workspace, the full name of the workspace would be SCOTT.DEMO. If you have the rights to access a workspace that resides in another user's schema, you can specify the full name when you attach the workspace. For example SCOTT could attach a workspace called REPORTS in a schema owned by SUSAN with the following command.

```
aw attach susan.reports
```

In almost any DML command, you can specify the full name of a workspace (for example, SCOTT.DEMO). If the workspace is in your schema, you can specify only the name (for example, DEMO) instead. Optionally, you can reference a workspace using an assigned workspace alias.

Workspace Aliases

A workspace alias is an alternative name for an attached workspace. You can assign or delete an alias with the AW ALIASLIST command.

An alias is in effect from the time it is assigned to the time that the workspace is detached (or until the alias is deleted). Therefore, each time you attach an unattached workspace, you must reassign its aliases.

One reason for assigning an alias is to have a short way to reference a workspace that belongs to a schema that is not yours. For example, you can use the alias in qualified object names and commands that reference such a workspace. Another reason for assigning an alias is to write generic code that includes a reference to a workspace but does not hard-code its name. With the alias providing a generic reference, you can assign the alias and run the code on different workspaces at different times.

Saving Analytic Workspace Changes

Typically, you want to save an analytic workspace at the end of your session to save changes that were made during the session. You can also save an analytic workspace periodically during a session to save changes as you go along.

If you have read/write access to the analytic workspace, then you can save the changes you have made. If you have read-only access to the analytic workspace, then you can make changes to the analytic workspace, but you cannot save these changes.

Two commands are used together to save changes to an analytic workspace: UPDATE and COMMIT.

UPDATE Command

The UPDATE command moves analytic workspace changes from a temporary area to the database table in which the workspace is stored. Your changes are not saved until you execute a COMMIT command, either from the OLAP DML or from SQL.

If you want changes that you have made in a workspace to be committed when you execute a COMMIT command, then you must first update the workspace using the UPDATE command. Changes that have not been moved to the table are not committed.

The simplified syntax for the UPDATE command is show below.

```
UPDATE [awname1 [awname2 . . .]]
```

An *awname* argument specifies the name of a read/write analytic workspace that is attached to your session. If you do not specify any analytic workspace names, then all the attached read/write analytic workspaces are updated.

For example, you can issue the following command to move changes to all attached analytic workspace from a temporary area to the database tables in which the workspaces are stored.

```
UPDATE
```

COMMIT Command

The COMMIT command executes a SQL COMMIT command. All changes made during your session are committed, whether they were made through Oracle OLAP or through another form of access (such as SQL) to the database.

For example, you can issue the following two commands to save all analytic workspace changes in the database.

```
UPDATE
COMMIT
```

Many users execute DML commands using SQL*Plus or OLAP Worksheet. Both of these tools automatically execute a COMMIT command when you end your session. However, you must first execute an UPDATE command in order to save your analytic workspace changes.

If you have attached a shared analytic workspace and another user has read/write access, then that user's COMMIT command does not affect your view of the analytic workspace. Your view of the data remains the same as when you attached the

analytic workspace. If you want access to the changes, then you must detach the analytic workspace and reattach it.

Effect of the ROLLBACK Command

The OLAP DML does not provide a way to issue a `ROLLBACK` command; however, you could execute one in your session from outside Oracle OLAP (for example, through PL/SQL). When a `ROLLBACK` command is executed in your session, Oracle OLAP checks to see whether there are uncommitted updates in an attached workspace.

- If there are uncommitted updates (that is, you have made changes and executed an `UPDATE` command, but you have not subsequently executed a `COMMIT` command), then Oracle OLAP discards your changes and detaches the workspace.
- If you have no uncommitted updates, then Oracle OLAP takes no action in response to the `ROLLBACK` command. This means that, if you have not issued an `UPDATE` command since your last `COMMIT` command, Oracle OLAP takes no action and all your changes remain in the workspace during your session.

If you rollback to a savepoint and there are uncommitted updates that occurred subsequent to the savepoint, Oracle OLAP discards those updates and detaches the workspace. Uncommitted updates that occurred before the savepoint remain in the workspace, and you can see them if you reattach the workspace in the same session.

Minimizing Analytic Workspace Growth

You can minimize analytic workspace growth by frequently updating the analytic workspace when you are attached exclusively. You can reorganize your analytic workspace files by exporting all of the objects in your analytic workspace and then importing them into a new analytic workspace. The new workspace may be substantially smaller.

To reorganize your analytic workspace by exporting and importing workspace objects, follow the procedure outlined below.

1. Issue an `ALLSTAT` command against the original analytic workspace.
2. Use the `EXPORT` command with the `ALL` keyword to put all of the data in the original analytic workspace into an EIF file.
3. Create a new analytic workspace with a different name than the original analytic workspace.

4. Use the `IMPORT` command to import the EIF file into the new analytic workspace.
5. Use the `UPDATE` and `COMMIT` commands to save the new analytic workspace.
6. After checking that the objects were successfully moved into the new analytic workspace, delete the original analytic workspace.

If you have programs that reference a given workspace, they can refer to the workspace by an alias. This way, it does not matter how many times you import to a workspace with a different name. The alias can be assigned to the appropriate workspace each time.

Executing Programs Automatically

You can create programs that are automatically executed when you attach an analytic workspace. When you attach an analytic workspace by using the `AW ATTACH` command with the `AUTOGO` keyword, the workspace is searched for a program named `AUTOGO`. If it exists, then the program is executed before commands are accepted. If you do not specify the `AUTOGO` keyword, or if you specify the `NOAUTOGO` keyword, the program is not automatically executed.

Program Names

If you have a program named `AUTOGO` in more than one currently attached analytic workspace (and thus multiple programs with the same name), then you must use their qualified object names when you edit them to ensure that you are accessing the correct one.

You do not have to name a program `AUTOGO` to have it automatically execute when you specify the `AUTOGO` keyword. Instead, you can use the `AUTOGO` keyword with the name of the program that you want executed. Even if a program named `AUTOGO` exists in the analytic workspace, Oracle OLAP executes the program you specify with the `AUTOGO` keyword.

AUTOGO Program Example

Suppose you have two analytic workspaces of sales data, one for expenses and one for revenue. You have a third analytic workspace called `analysis` that contains programs that analyze the data. In the `analysis` workspace, you can have an `AUTOGO` program that includes the following two lines of code for attaching the other two workspaces.

```
AW ATTACH expense AFTER analysis
AW ATTACH revenue AFTER analysis
```

When you attach the `analysis` workspace with the following command, its `AUTOGO` program runs automatically and attaches the other two workspaces.

```
AW ATTACH analysis AUTOGO
```

If you named the program `ATTACHDATA` instead of `AUTOGO`, you would attach the `analysis` workspace with the following command.

```
AW ATTACH analysis AUTOGO attachdata
```

Note that permission programs are executed before any `AUTOGO` program is executed.

See Also:

- "Adding Security to an Analytic Workspace" on page 2-12 for information about permission programs.
- Chapter 7, "Developing Programs" for information on writing programs.

Adding Security to an Analytic Workspace

An analytic workspace as an entity is protected with all of the security features built into the database. In addition, you can restrict access to specific workspace objects, or to an entire workspace, with permission programs.

Permission Programs

When a user attaches an analytic workspace, it is checked to see if it contains permission programs, which are called `permit_read` and `permit_write`. You do not have to create these programs; however, if they are present, then the appropriate one is automatically executed when a user attaches the analytic workspace.

IF the user attaches an analytic workspace with . . .	THEN the following program is executed, if it exists . . .
read-only access,	`permit_read` program.
read/write access,	`permit_write` program.

Permission programs are executed before any AUTOGO program is executed. If a user specifies a password when attaching the analytic workspace, then the password is passed as an argument to the permission program for processing. The permission program can grant or restrict access to the entire workspace or to individual objects based on the password that has been provided. For example, in the following AW command, the sales workspace is attached with goldfinch as the password.

```
AW ATTACH sales PASSWORD goldfinch
```

Creating and Designing Permission Programs

To create permission programs, you define two programs with the names permit_read and permit_write. In these programs, you can specify PERMIT commands that grant or restrict access to individual workspace objects. In addition, you write these programs as user-defined functions that return a Boolean value, and the return value indicates to Oracle OLAP whether or not the user has the right to attach the workspace.

IF the program returns . . .	THEN the analytic workspace . . .
YES	is attached.
NO	is not attached.

Thus, permission programs allow you to control two levels of access to the analytic workspace in which they reside.

Type of access	Description
Analytic workspace level	Depending on the return value of the permission program, the user is or is not granted access to the entire analytic workspace.
Object level	Depending on the PERMIT commands in the permission program, the user is granted or denied access to specific objects or sets of object values. All of the objects referred to in a given permission program must exist in the same analytic workspace.

For example, using the PERMIT command, you can deny access to the salary variable to one group of users, and you can deny access to the tenure variable to another group of users. You can even specify that certain users cannot access a subset of the cells in the salary variable.If you have permission programs in more than one currently attached analytic workspace (and thus, multiple programs with

the same name), then you must use their qualified object names when you edit them, to ensure that you are accessing the correct one.

> **See Also:** Chapter 7, "Developing Programs" for information on writing programs.

Importing and Exporting Workspace Objects

You can export an entire workspace, several workspace objects, a single workspace object, or a portion of a workspace object to a specially formatted EIF file. Then you can import the information into a different workspace within the same Oracle database or a different one.

One reason for exporting and importing is to move your data to a new location. Another purpose is to remove extra space from your analytic workspace after you have added and then deleted many objects or dimension values. To do this, use the `EXPORT` command to put all the data in an EIF file, create another workspace with a different name, and then use the `IMPORT` command to import the EIF file into the new workspace. If you have imported into the same database, you can delete the old workspace and refer to the new one with the same workspace alias that you used for the original one.

The following command copies all the data and definitions from the current analytic workspace to an EIF file called `reorg.eif` in a directory alias called `mydir`.

```
export all to eif file 'mydir/reorg.eif'
```

Directory aliases are defined in the database, and they control access to directories. You can use the `CDA` command to specify a current directory alias. In this case, you do not have to specify a directory alias in the `EXPORT` command, because Oracle OLAP assumes that you want the file to be created in your current directory alias. Contact your Oracle DBA for access rights to a directory alias where your database user name can read and write files.

The following command copies the data and definitions from the EIF file to a new analytic workspace.

```
aw create new
import all from eif file 'reorg.eif'
update
commit
```

Obtaining Analytic Workspace Information

The AWDESCRIBE program displays a complete description of your analytic workspace, including:

- A table of contents that shows general information about your analytic workspace, such as the date and time of the last update and the number of each type of workspace object.
- A list of workspace objects that are sorted alphabetically.
- Detailed descriptions of all workspace objects, which are sorted by type of object and sorted alphabetically by name within each type. For each object, there is a cross-reference list of other objects that use or are used by this object.

Because the output from AWDESCRIBE is frequently very long, you can direct it to a file with the OUTFILE command:

```
OUTFILE 'diralias/filename'
AWDESCRIBE
OUTFILE EOF
```

Where ***diralias*** is the name of a directory alias, and ***filename*** is the name of the file where the information will be written.

Contact your DBA for the name of a directory alias to which you have read and write privileges.

Obtaining General Information About an Analytic Workspace

The AW function returns various kinds of information about attached analytic workspaces. For example, you can use the AW function to learn the name of your current workspace or whether you have read/write access to it.

The simplified syntax of the AW function is shown below.

```
AW(choice [workspace])
```

The keyword you specify for ***choice*** determines the type of information that is returned by the AW function. Examples of keywords are: ATTACHED, NAME, RO, and RW.

For example, the following commands check which analytic workspace is active so the program can choose the appropriate data to report.

```
IF AW(NAME) EQ 'mysales'
     THEN REPORT sales.m
     ELSE REPORT gensales
```

Viewing Objects in an Analytic Workspace

You can retrieve a list of the objects in an analytic workspace by using the `LISTNAMES` program. This program lists all the objects in the analytic workspace, grouped by object type and alphabetized within object type. `LISTNAMES` shows the total number of each type of object (dimension, variable, and so on).

Use the `LISTBY` command to retrieve a list of all objects that are dimensioned by, or related to, a given dimension.

For example, to find out which objects are dimensioned by, or related to, `month`, you can use the following command.

```
LISTBY month
```

The following list is displayed.

```
14 objects dimensioned by or related to MONTH in DEMO
--------------------------------------------------------------
ACTUAL         ADVERTISING        BUDGET
EXPENSE        FCST               NATIONAL.SALES
PRICE          PRODUCT.MEMO       SALES
SALES.FORECAST SALES.PLAN         SHARE
UNITS          UNITS.M
```

To display the definitions of one or more objects, use the `DESCRIBE` command. For example, you can issue the following command.

```
DESCRIBE price
```

It produces the following output.

```
DEFINE PRICE VARIABLE DECIMAL <MONTH PRODUCT>
LD Wholesale Unit Selling Price
```

If you execute the `DESCRIBE` command without any object names, all the objects in the current status list of the `NAME` dimension are described. The `NAME` dimension contains the names of all the objects that are defined in the analytic workspace.

You can display the values of many workspace objects, such as variables, dimensions and relations, by executing a REPORT command. For example, the following command shows the values of a variable called costs.

```
report costs
```

This command might produce the following output.

```
          ------------------------COSTS------------------------------
          ----------------------GEOGRAPHY----------------------------

DIVISION     EAST       WEST      BOSTON   SAN FRANCISCO      SEATTLE
--------- ---------- ---------- ---------- -------------    ----------
DIVA       27,600.00  50,000.00  27,600.00  10,000.00        40,000.00
DIVB       30,000.00  62,000.00  30,000.00  12,000.00        50,000.00
```

Obtaining Information About Objects

To obtain information about workspace objects, you can use the OBJ function.

For example, the following command obtains the number of dimensions for the variable units. The output is shown below the command.

```
SHOW OBJ(NUMDIMS 'units')
3
```

The following command obtains the data type of the units variable. The output is shown below the command.

```
show obj(data 'units')
INTEGER
```

You often use the OBJ function in conjunction with the LIMIT command and the NAME dimension in order to obtain information about groups of objects. The LIMIT command sets the status of a dimension. This means that it restricts the accessibility of dimension values, which sets a corresponding restriction on any variables or relations that are dimensioned by them.

You can use the LIMIT command together with the OBJ function to identify a group of objects with a particular characteristic. Then, you can list the objects in the group using the STATUS command.

The following commands lists the objects that are dimensioned by both `month` and `product`.

```
LIMIT NAME TO OBJ(ISBY 'month') AND OBJ(ISBY 'product')
STATUS NAME
```

The output of these commands is shown below.

```
The current status of NAME is:
ADVERTISING, EXPENSE, NATIONAL.SALES, PRICE, PRODUCT.MEMO, SALES,
SALES.FORECAST, SALES.PLAN, SHARE, UNITS, UNITS.M
```

See Also: Chapter 6, "Selecting Data" for information about using the `LIMIT` command.

3

Defining Data Objects

This chapter introduces multidimensional data structures. It explains how to define objects and change the definition of those objects. It includes the following topics:

- Overview: Defining Workspace Objects
- Data Types
- Defining Dimensions
- Defining Relations
- Defining Variables
- Defining Variables That Handle Sparse Data Efficiently
- Defining Hierarchical Dimensions and Variables That Use Them
- Defining Concat Dimensions and Variables That Use Them
- Changing the Definition of an Object

Overview: Defining Workspace Objects

It is important to understand the distinction between the definition of an object and its data. An object definition is its description in the analytic workspace. The data of an object is the value or values that are associated with that definition. All objects have definitions. However, not all objects have data.

For example, a `sales` variable that is dimensioned by `month`, `product`, and `district` has a definition for itself as a variable object. The `sales` variable is also associated with the definitions for its three dimensions. However, the values of `sales`, `month`, `product`, and `district` are not part of the definitions.

Other objects, such as programs and formulas, do not have data.

Once you have created an analytic workspace, you can begin defining workspace objects. To define any OLAP DML object, use the `DEFINE` command. The simplified syntax for the `DEFINE` command is shown below.

```
DEFINE name object-type attributes
```

The *`name`* argument specifies the name for the new definition.

Note: Because each analytic workspace has its own list of workspace objects, you can define objects with the same name in more than one analytic workspace. However, to prevent unexpected results, you should provide unique names for objects in separate analytic workspaces that will be active at the same time, unless you are prepared to use qualified object names as described in Chapter 2, "Defining and Working with Analytic Workspaces".

The *`object-type`* argument specifies the type of object that is being defined. The default is `VARIABLE`. You can specify any of the valid object types as outlined in "Workspace Objects That You Can Define" on page 3-3.

The *`attributes`* argument specifies the properties of the object. Attributes are different for each type of object. The attributes are listed in the entry for each object type.

Workspace Objects That You Can Define

The OLAP DML data object types that you define using the `DEFINE` command are outlined in the following table.

Object Type	Description
`DIMENSION`	Contains a list of values that provide categories for data. A dimension acts as an index for identifying values of a variable. A dimension is similar to a key in a relational database.
`RELATION`	Establishes a correspondence between the values of a given dimension and the values of that dimension or other dimensions in the analytic workspace. A relation is similar to a foreign key in a relational database.
`VARIABLE`	Stores data. The data type of a variable indicates the kind of data that it contains. A variable is similar to a table in a relational database.
`COMPOSITE`	A named list of dimension-value combinations, in which a given combination has one value taken from each of the dimensions on which the composite is based. **Note:** An unnamed composite is automatically created when you define a variable with some dimensions specified as sparse. An unnamed composite is an internal object; it is not considered an OLAP DML object.
`SURROGATE`	Contains a list of values that are surrogates for the values of a simple dimension. You can use a surrogate for a dimension in `LIMIT` commands, models, qualified data references, and data loading.
`FORMULA`	Represents a stored calculation, expression, or procedure that produces a value. A formula is similar to a view in a relational database.
`MODEL`	Contains a set of interrelated equations that are used to calculate data and assign it to a variable or dimension value. In most cases, models are used when working with financial data.
`PROGRAM`	Contains a series of OLAP DML commands. A program executes a set of related commands. A program is similar to a SQL stored procedure.
`VALUESET`	Contains a list of dimension values for a particular dimension.

Object Type	Description
AGGMAP	Creates an aggregation map, which can contain commands that specify which data in a variable should be aggregated or allocated and how the operation is performed. With the AGGMAP command, you can specify commands used by the AGGREGATE command. With the ALLOCMAP command, you can specify commands used by the ALLOCATE command.

Data Types

Workspace data types fall into categories, which are referred to as basic data types. They are listed in the following table.

Basic Type	Specific Type
Numeric	INTEGER, SHORTINTEGER, LONGINTEGER, DECIMAL, SHORTDECIMAL, NUMBER
Text	TEXT, NTEXT, ID
Boolean	BOOLEAN
Date	DATETIME, DATE

Different objects support the use of different data types for their values:

- For most data values, such as those stored in variables, the INTEGER, SHORTINTEGER, DECIMAL, SHORTDECIMAL, NUMBER, TEXT, ID, NTEXT, BOOLEAN, DATETIME, and DATE data types are supported.
- For dimension values, the INTEGER, NUMBER, TEXT, ID, and NTEXT data types are supported.

Numeric Data Types

The following numeric data types are supported.

Data Type	Data Value
INTEGER	A whole number in the range of (-2**31) to (2**31)-1.
SHORTINTEGER	A whole number in the range of (-2**15) to (2**15)-1.
LONGINTEGER	A whole number in the range of (-2**63) to (2**63)-1.
DECIMAL	A decimal number with up to 15 significant digits.

Data Type	Data Value
SHORTDECIMAL	A decimal number with up to 7 significant digits.
NUMBER	A decimal number with up to 38 significant digits.

For data entry, a value for any of these data types can begin with a plus (+) or minus (-) sign; it cannot contain commas. Additionally, a decimal value can contain a decimal point. For data display, thousands and decimal markers are controlled by the NLS_NUMERIC_CHARACTERS option.

The workspace NUMBER data type is fully compatible with the database NUMBER data type. It is used for dimensions and surrogates when a text or integer data type is not appropriate. It is typically assigned to variables that are not used for calculations (like forecasts and aggregations), and it is used for variables that must match the rounding behavior of the database or require a high degree of precision. When deciding whether to assign the NUMBER data type to a variable, keep the following facts in mind in order to maximize performance:

- Analytic workspace calculations on NUMBER variables is slower than calculations on other numeric types such as DECIMAL.
- When data is fetched from an analytic workspace to a relational column that has the NUMBER data type, performance is best if the data already has the NUMBER data type in the analytic workspace because a conversion step is not required.

Examples of Literal Numeric Values

Examples of literal numeric values are:

```
-1
256000
+2147483647
10000000000.0009
```

Text Data Types

The following text data types are supported.

Data Type	Data Value
TEXT	Up to 4000 bytes per line in the database character set. This data type is equivalent to the CHAR and VARCHAR2 data types in the database.

Data Type	Data Value
NTEXT	Up to 4000 bytes per line in UTF-8 character encoding. This data type is equivalent to the NCHAR and NVARCHAR2 data types in the database.
ID	Up to 8 characters per line in the database character set

For data entry, text literals must be enclosed in single quotes. Otherwise, the OLAP DML command processor will look for a workspace object by that name.

Escape Sequences

In some cases, text data includes values that are not printable. Escape sequences are provided to allow such values to be input and displayed. An escape sequence is a series of alphanumeric characters that begins with a backslash.

The following table shows escape sequences that are recognized.

Escape Sequence	Meaning
\b	Backspace
\f	Form feed
\n	Line feed
\r	Carriage return
\t	Horizontal tab
\"	Double quote
\'	Single quote
\\	Backslash
\d*nnn*	Character with ASCII code *nnn* decimal, where \d indicates a decimal escape and *nnn* is the decimal value for the character
\x*nn*	Character with ASCII code *nn* hexadecimal, where \x indicates a hexadecimal escape and *nn* is the hexadecimal value for the character
\U*nnnn*	Character with Unicode *nnnn*, where \U indicates a Unicode escape and *nnnn* is a four-digit hexadecimal integer that represents the Unicode codepoint with the value U+*nnnn*. The U must be a capital letter.

Examples of Literal Text Values

Examples of literal text values are:

```
'Raoul D\'Allesandro'
'NONE'
'January 2002'
```

Boolean Data Type

A Boolean data type is provided that you can use to represent logical values. In code, you can use any of the following values (in any combination of uppercase and lowercase characters) to represent Boolean values:

- YES, TRUE, ON
- NO, FALSE, OFF

The values that are used in your installation are determined by the language identified by the NLS_LANGUAGE option. You can use the read-only NOSPELL and YESSPELL options to obtain the values.

Working with Boolean expressions is discussed in "Boolean Expressions" on page 4-21.

Date Data Types

The following date data types are supported.

Data Type	Data Value
DATETIME	Dates between January 1, 4712 B.C. and December 31, 9999 A.D., and times in hours, minutes and seconds.
DATE	Dates between January 1, 1000 A.D. and December 31, 9999 A.D.

The format and language of DATETIME values are controlled by the NLS_DATE_FORMAT and NLS_DATE_LANGUAGE options. The DATETIME data type is supported by Oracle standard libraries and operates the same way in the database, and thus is preferable to the DATE data type. The DATEORDER, DATEFORMAT, and MONTHNAMES options, which control the formatting of DATE values, have no effect on DATETIME values. However, DATETIME and DATE values can be used interchangeably in most DML commands

Defining Dimensions

A dimension is an object that holds a list of values that provide the organization for one or more variables. A dimension value is similar to a key in a relational table; either alone or with other dimension values, it uniquely identifies a data value. For example, if you have sales data with a separate sales figure for each month, then the data has a `month` dimension; that is, the data is organized by month. The dimension values you add might be `feb02`, `mar02`, and `apr02`.

A **simple dimension** has a list of values that all have the same data type. The OLAP DML supports both flat and hierarchical simple dimensions:

- A **flat dimension** exists when the values within a dimension are all at the same level. No value is the child or parent of another value.
- A **hierarchical dimension** exists when values are in a one-to-many (parent-to-child) relationship with each other. A hierarchical dimension is a means of organizing and structuring this type of data within a single dimension. You can then use it to dimension a variable that contains data for all the levels. Some dimensions have multiple hierarchies. You specify the parent-to-child relationships of the dimension values by creating a self-relation.

Composite and conjoint dimensions can be derived from these base simple dimensions to store sparse data more efficiently in a multidimensional format.

Concat dimensions can also be based on simple dimensions or on conjoint dimensions. You can represent a hierarchy with a concat dimension that is has two or more simple flat dimensions among its base dimensions. You can use concat dimensions to easily map dimensions in an analytic workspace to columns in relational tables and thereby promote more efficient loading of data from the relational structures into the analytic workspace structures. The base dimensions of a concat dimension can be of different data types.

See Also:

- "Defining Variables That Handle Sparse Data Efficiently" on page 3-18 for information about composite and conjoint dimensions.
- "Defining Hierarchical Dimensions and Variables That Use Them" on page 3-22 for information about hierarchical dimensions.
- "Defining Concat Dimensions and Variables That Use Them" on on page 3-25 for information about concat dimensions.

Determining What Dimensions to Define

If you want your analytic workspace to contain only flat dimensions, you need to define dimensions for each level of detail in your data that users will access.

For example, if your company is divided into sales districts and each district handles several store accounts, then you need to decide whether you want sales figures for every store or only for each district. As shown in the following table, the answer to this question determines the structure of your analytic workspace.

IF . . .	THEN . . .
you need Store data,	you can define a `store` dimension.
you always look at each district as a whole,	all you need is a `district` dimension.
you want to look at data both ways,	you can organize data by store and view aggregates of data by district by creating both a `store` and a `district` dimension with a relation between them.

Sometimes, you will decide to store data of varying levels of aggregation within a single variable, because this type of storage affords a quicker response time for users who want to view the data. In this case, you can define a single hierarchical dimension that has all the values of the hierarchy or you can define a concat dimension that is based on simple flat dimensions. For example, each flat dimension might have the values of one of the levels of the hierarchy.

For example, if you want to look at data both ways instead of defining both a `store` and a `district` dimension as described above, then you can define a single hierarchical dimension. This hierarchical dimension would contain all of the values

for stores and districts. If you dimension a variable by this hierarchical dimension, then you can store data of varying levels of aggregation within that single variable. You can still view store data and district data separately.

You can achieve a similar result by defining a concat dimension that has as its base dimensions the `store` and `district` dimensions. The concat dimension would also contain all of the store and district values. As with a hierarchical dimension, if you dimension a variable by this concat dimension, then you can keep data of varying levels of aggregation within that variable and still view store data and district data separately.

If you already have simple flat dimensions in your analytic workspace or if you create simple flat dimensions so that you can easily map them to columns in relational dimensions, then you might use a concat dimension instead of a hierarchical dimension. Another reason to use a concat dimension instead of a hierarchical simple dimension is that all of the values of a simple dimension must be unique whereas in a concat dimension the same value can exist in two or more of the base dimensions of the concat.

How Data For Simple Flat Dimensions Is Stored

The data for a simple flat dimension is stored in a one-dimensional array. As you add values to the dimension, each new value is stored at the end of the array.

Assume that the `product` dimension has been defined as a `TEXT` data type. The first three values that are added to the dimension are `TENTS`, `CANOES`, and `RACQUETS`. At this point, a report of the dimension shows the following values.

```
PRODUCT
--------------
TENTS
CANOES
RACQUETS
```

The `product` dimensionvalues are actually stored as shown below.

Position	**1**	**2**	**3**
Value	TENTS	CANOES	RACQUETS

Later, the values SPORTSWEAR and FOOTWEAR are added. At this point, a report of the dimension shows the following values.

```
PRODUCT
--------------
TENTS
CANOES
RACQUETS
SPORTSWEAR
FOOTWEAR
```

Now the product dimension array looks like the following.

Position	**1**	**2**	**3**	**4**	**5**
Value	TENTS	CANOES	RACQUETS	SPORTSWEAR	FOOTWEAR

See Also: Chapter 5, "Populating Workspace Data Objects", for information about adding dimension values.

Defining Dimension Surrogates

A dimension surrogate is an object that provides an alternative way to specify the positions of a dimension. As described in "How Data For Simple Flat Dimensions Is Stored", on page 3-10 each value of a dimension is identified by a position in the dimension. The position is specified by an integer. For an INTEGER type dimension, the values and the positions are the same. In a LIMIT command or a qualified data reference (QDR) you can use the value of the dimension or the position of the value in the dimension. For example, the following commands both set the status of the product dimension to the same value.

```
LIMIT product TO 'TENTS'
LIMIT product TO 1
```

A primary key column in a relational table might have values that are numbers. To efficiently load data from the relational structures into your analytic workspace, you can define a NUMBER dimension to contain the primary key values. NUMBER dimensions are different than other types of dimension because you cannot specify a value of a NUMBER dimension by its position in the dimension. However, you can define an INTEGER type dimension surrogate for the NUMBER dimension and use

the values of the surrogate in LIMIT commands, models, QDRs, and data loading instead of using the primary key values from the NUMBER dimension.

You can define dimension surrogates for simple dimensions and for conjoints but not for concat dimensions or composites. For example, you might want to have a conjoint dimension but also want to have a single text value to specify each value of the conjoint. You can accomplish that by creating a TEXT dimension surrogate for the conjoint dimension. If you define a dimension surrogate for a conjoint dimension, then you cannot convert the conjoint dimension to a composite dimension.

You can define any number of dimension surrogates for a dimension. The type of the dimension surrogate does not have to be the same as the type of the dimension. You can define a dimension surrogate for any type of dimension other than the time types DAY, WEEK, MONTH, QUARTER, or YEAR. However, these time types are provided only for compatibilty with earlier versions. Using them is not currently recommended.

Differences Between Dimensions and Dimension Surrogates

You cannot dimension an object by a dimension surrogate. However, you can dimension an object, such as a variable, by a dimension, define a dimension surrogate for the dimension, and then use the values of the surrogate instead of the dimension in LIMIT commands, models, QDRs, and data loading.

You cannot define a valueset on a dimension surrogate. However, you can define a valueset on a dimension, define a dimension surrogate for the dimension, and then specify values for the valueset by using values of the surrogate in a LIMIT command.

You cannot define a relation on a dimension surrogate. However, you can define dimension surrogates for the dimensions that dimension a relation and then use the values of the surrogates in LIMIT commands or QDRs.

You cannot use a surrogate as the data type of a program or a formula.

You cannot add new positions directly to a dimension surrogate. However, with the MAINTAIN command you can add values to the dimension on which you have defined the surrogate. The surrogate then automatically has a new position for each value you that add to the dimension.

A dimension surrogate does not have its own status. It shares the status of its dimension. You can uses the values or positions of a dimension surrogate or its dimension with a LIMIT command or a QDR to set the status of the dimension and the dimension surrogate.

You cannot delete a dimension if a dimension surrogate exists for that dimension. However, you can delete the dimension surrogate without affecting the dimension.

You cannot use the `PERMIT` command on a dimension surrogate. A surrogate has the permissions set on its dimension.

You cannot use a dimension surrogate in commands that use the `ACROSS` or `DOWN` keywords to loop over, total over, or report over specified dimensions. In those cases, you must specify the dimension and not a surrogate for it.

You cannot use the `CHDGFN` or `MAINTAIN` commands on a dimension surrogate. However, you can use dimension surrogate values in a `MAINTAIN` command to specify values for a dimension.

Defining Relations

A relation is an object that establishes a correspondence between the values of a given dimension and the values of that same dimension or other dimensions in the analytic workspace. The structure of a relation is similar to that of a variable. However, the cells in relations do not hold actual data values; instead, each cell in a relation holds the index of the value of a dimension.

By creating a relation between two dimensions that participate in a one-to-many (parent-to-child) relationship, you can organize your data by the child dimension and view aggregates of data by the parent dimension. For example, if you define `store` and `district` dimensions and a relation between them, then you can organize data by `store` and view aggregates of data by `district`.

You can explicitly define relations between two or more dimensions, multiple relations between a set of dimensions, or a dimension with itself (a **self-relation**).

How Relations Are Dimensioned

Relations are dimensioned arrays. Relations can be dimensioned by the dimension with the larger number of values or the smaller number of values.

Typically, a relation is dimensioned by the dimension with the larger number of values (that is, the less aggregate or child dimension) and the related dimension is the dimension with fewer values (that is, the more aggregate or parent dimension). For example, you can create a relation called `state.city` to associate each city with the state that it is in. The relationship is dimensioned by `city` and the related dimension is `state`. You assign a state to each city.

Less typically, a relation is dimensioned by the dimension with fewer values (the more aggregate dimension or parent dimension). In this case, not every value of the other dimension is related. For example, you could create a relationship, named `city.state`, between states and their capital cities. The relation is dimensioned by `state` and the related dimension is `city`. Only the capital cities are assigned to a state.

How Relation Data Is Stored

The order in which you define the dimensions of a relation determines how its data is stored and accessed. Dimensions vary in the order you list them in the definition, with the first dimension varying fastest and the last dimension varying slowest. See "How Variable Data Is Stored" on page 3-17 for information on faster- and slower-varying dimensions.

The data values that are stored for a relation are the indexes of the related dimension. The index is the position of the value in the dimension.

For example, the `state.city` relation (that is dimensioned by `city` and has a related dimension of `state`) assigns a state to each city. To implement this relationship, an index from the `state` dimension is stored for every value (index) in the `city` dimension. The following table shows the positions of the `city` dimension that are assigned to each position of the `state` dimension. It also shows the values at those positions in the dimensions.

City Position (Index)	City Value at Position	State Position (Index)	State Value at Position
1	Atlanta	1	Georgia
2	Chicago	2	Illinois
3	Springfield	2	Illinois

See Also:

- Chapter 5, "Populating Workspace Data Objects" for information about adding values to relations.
- Chapter 4, "Working with Expressions" for information about using relations in expressions.

Example: Relation Between Two Dimensions

Most relations are a single-dimensional array that relates the values of one dimension with another. For example, you can define two simple dimensions, `state` and `city`, and a relation `state.city` between them to associate each city with the state that it is in.

Assume that the `state.city` relation was defined using the following command.

```
DEFINE state.city RELATION state <city>
```

Assume that, as shown below, the `state` dimension has two values and the `city` dimensions has three values.

```
STATE
--------------
GEORGIA
ILLINOIS

CITY
--------------
ATLANTA
CHICAGO
SPRINGFIELD
```

The `state.city` relation is dimensioned by `city` and the related dimension is `state`. The `state.city` relation assigns a state to each city as shown below.

```
CITY           STATE.CITY
-------------- ---------------
ATLANTA        GEORGIA
CHICAGO        ILLINOIS
SPRINGFIELD    ILLINOIS
```

Example: Self-relation

You can define a self-relation for a single dimension. For example, to keep track of the reporting structure of a company, you can have the `emp.emp` relation for the `employee` dimension.

Assume that the `emp.emp` relation was defined using the following command.

```
DEFINE emp.emp RELATION employee <employee>
```

Assume that the `employee` dimension contains the values shown below.

```
EMPLOYEE
--------------
ANN LOGAN
MICHAEL ARON
LUCY BATES
RALPH BURNS
```

The self-relation `emp.emp` is dimensioned by the `employee` dimension and the related dimension is also the `employee` dimension. As shown below, the `emp.emp` relation assigns a manager to each employee.

```
EMPLOYEE        EMP.EMP
-------------- ----------
ANN LOGAN      NA
MICHAEL ARON   ANN LOGAN
LUCY BATES     ANN LOGAN
RALPH BURNS    LUCY BATES
```

In this example, Ann Logan, the company president, does not report to anyone; employees Lucy Bates and Michael Aron report directly to Ann Logan, the president; and employee Ralph Burns reports to employee Lucy Bates.

See Also:

- "Defining Hierarchical Dimensions and Variables That Use Them" on page 3-22 for information about using self-relations with hierarchical dimensions.
- "Defining Concat Dimensions and Variables That Use Them" on on page 3-25 for information about using self-relations with concat dimensions.

Defining Variables

A variable is an object that stores data. All of the data in a variable represents the same unit of measurement with the same data type. Your business might have several categories of transactions (measured in dollars, units, percentages, and so on) and each category is stored in its own variable. For example, you might record sales data in dollars (a `sales` variable) and units (a `units` variable).

Typically, you use variables to contain data values that quantify a particular aspect of your business.

Types of Variables

Variables can be either dimensioned or undimensioned:

- Dimensioned variables. If a variable is an array with dimensions, then those dimensions organize its data, and there is one cell for each combination of dimension values. This type of variable is called a **dimensioned variable**. A variable can be dimensioned by up to 32 dimensions.
- Undimensioned variables. If a variable has no dimensions, then it is a **scalar**, or single-cell variable, which contains one data value.

Variables that you define in an analytic workspace can be permanent or temporary. You can also define variables in programs, as described in "Defining Local Variables" on page 7-6.

A permanent variable is a variable for which both the variable values and definitions are stored in an analytic workspace.

Temporary variables have values only during the current session. When you update and commit the analytic workspace, only the definitions of temporary variables are saved. When you exit from the analytic workspace, the data values are discarded.

How Variable Data Is Stored

The order in which you list the dimensions in a variable definition determines how the data of that variable is stored and accessed. The first dimension in the variable definition is the **fastest-varying dimension**, and the last dimension is the **slowest-varying dimension**.

For example, assume your analytic workspace has an `opcosts` variable that contains the operating costs, by month, of each city in which you have offices. In the definition shown below for the `opcosts` variable, `month` is the fastest-varying dimension and `city` is the slowest-varying dimension.

```
DEFINE opcosts VARIABLE DECIMAL <month city>
```

The data for a multidimensional variable is stored as a linear stream of values, in which the values of the fastest-varying dimension are clustered together. For example, for the `opcosts` variable, the values for Boston for all the months are stored in a sequence, and then it stores the values for Chicago for all the months in a sequence, and so on. Thus the month values vary fastest in the `opcosts` variable, as shown in the following table.

Dimension Values	JAN97 BOSTON	FEB97 BOSTON	. . . . . .	DEC97 BOSTON	JAN97 CHICAGO	. . . . . .
Variable Values	16000.77	16000.28	. . .	16000.98	19000.24	. . .

When you define variables and other dimensioned objects, and when you write programs that loop over multidimensional expressions in nested loops, you should always try to maximize performance by matching the fastest-varying dimension with the inner loop.

Defining Variables That Handle Sparse Data Efficiently

A variable with sparse data is one in which a relatively high percentage of the cells of the variable do not contain actual data. Such “empty,” or NA, values take up storage space in the analytic workspace.

There are two types of sparsity:

- **Controlled sparsity** occurs when a range of values of one or more dimensions has no data; for example, a new variable dimensioned by `month` for which you do not have data for past months. The cells exist because you have defined past months in the `month` dimension, but the cells are empty.
- **Random sparsity** occurs when NA values are scattered throughout the data variable, usually because some combinations of dimension values never have any data. For example, a district might only sell certain products and never have data for other products. Other districts might sell a different selection of products.

Definition: Composite

A **composite** is an internal object that is used to store sparse data compactly in a variable. A composite is a list of dimension-value combinations in which one value is taken from each of the dimensions on which the composite is based.

Composites can be named or unnamed:

- An **unnamed composite** is not a workspace object; it is merely an internal structure. When you define a variable, you use the `SPARSE` keyword to request that an unnamed composite is automatically created.

- A **named composite** is an object that is you define using the `DEFINE COMPOSITE` command. Later, when you are defining or accessing a variable, you can specify this composite by name along with the names of other dimensions.

Because the values in composites are maintained automatically, using composites is the recommended way of handling sparsity in your analytic workspace.

Using composites is one of the most important steps you can take to manage sparsity, which contributes to keeping analytic workspace size to a minimum and promoting good performance.

Why You Should Use Named Composites

Using named composites makes it easier to track which variables share the same composite. A named composite in the dimension list of a variable tells Oracle OLAP that the dimensions in the named composite are sparse in this variable, and that this composite is shared only with other variables that have the same sparsity pattern.

In contrast, all variables defined with an unnamed composite that have exactly the same dimensions in the same order will automatically share that unnamed composite. If these variables have different sparsity patterns, performance will suffer.

You can also manage sparsity by using a conjoint dimension to hold dimension-value combinations for which a given variable has data. However, because the values in composites are automatically maintained, using composites is the recommended way of handling sparsity in your analytic workspace.

How to Use Composites

When you define a multidimensional variable, you can specify a composite in the list of dimensions.

First, define a named composite by using the `DEFINE COMPOSITE` command. Then, define the variables by using the following syntax to include a named composite in the dimension list of each variable.

```
composite-name <dims>
```

For example, suppose you define a composite named `proddist`, whose dimensions include `product` and `district`, as shown in the following command.

```
DEFINE proddist COMPOSITE <product district>
```

Now, suppose you want to define a `sales` variable in which `time` will be the fastest-varying dimension and the `proddist` composite will be the slowest-varying dimension, as shown in the following command.

```
DEFINE sales <time proddist<product district>>
```

Note that you should never use the `SPARSE` keyword with a composite. Essentially, you use the name of the composite *instead of* the `SPARSE` keyword.

See Also:

- "Using Composites in Expressions" on page 4-13 for more information about using composites.
- "Working with NA Values" on page 4-32 for more information about working with sparse data.

Naming, Renaming, and Unnaming Composites

You can use the `RENAME` command to:

- Name an unnamed composite.
- Change the name of a named composite.
- Change a named composite to an unnamed composite.

Adding Data to a Variable That Uses a Composite

When you define a multidimensional variable, you can specify that a composite is used instead of its base dimensions to dimension the data. Later, as you add values to the dimensions of the variable for which you defined a composite, the following actions are taken:

- The composite is filled with dimension-value combinations.
- The data for the variable is stored using the composite structure rather than the structure of the base dimensions.

For a variable that uses a composite, cells are created for only those dimension values that are used in the composite dimension-value combinations; it does not create a variable cell for every value in the base dimensions. Data for a variable is stored in order, cell by cell, for each combination of dimension values. From the perspective of data storage, each combination of base dimension values in a composite is treated like the value of a regular dimension. This means that if you

define a variable with one regular dimension and one composite, then it is stored like a two-dimensional variable.

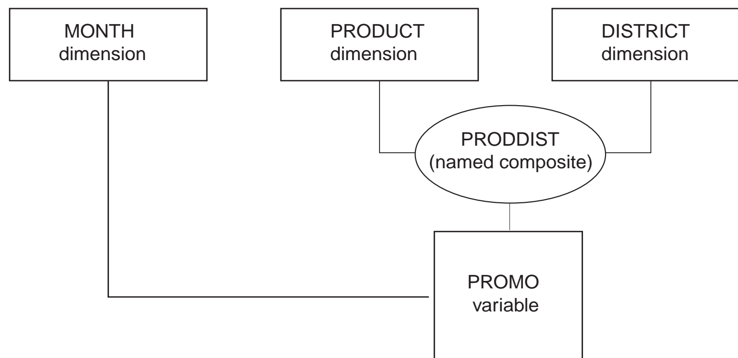
Example 3–1 Defining a Variable That Uses a Named Composite

If your company does promotional marketing for certain products in some but not all districts, then your variable data will be sparse along the `product` and `district` dimensions. Therefore, suppose you define a composite named `proddist`, whose base dimensions are `product` and `district`. There are dimension-value combinations in the composite only for those values that have data. For example, if you run a promotion for tents but not canoes, then the composite includes the tents and city combinations, but not the canoes and city combinations.

The following command creates a variable called `promo` that is dimensioned by `month` and a composite named `proddist`, whose base dimensions are `product` and `district`.

```
DEFINE promo INTEGER <month proddist<product district>>
```

The following conceptual figure illustrates the `promo` variable that is created by this command, the `month`, `product` and `district` base dimensions, a named composite (`proddist`) created from the `product` and `district` base dimensions, and the internal relation that is created between the `product` and `district` base dimensions and the `proddist` composite.



The following table is an example of the sequence in which the data for the promo variable might be stored.

TENTS BOSTON JAN95	TENTS BOSTON FEB95	TENTS BOSTON MAR95	. . . .	RACQUETS CHICAGO JAN95	RACQUETS CHICAGO FEB95	. . . .
257	379	428	. . .	635	192	. . .

Defining a Variable with a Single-Dimension Composite

When you specify a composite for just one dimension in a variable definition, a single-dimension composite is created. The values of this composite will be a subset of the values in its base dimension.

It is a good idea to use single dimension composites when a variable will share the same dimensions as some other variables, but for a particular single dimension, the variable will only have data for some of the values of the dimension.

Suppose you have already defined a variable called `actual` with the dimensions `line`, `division`, and `month`. The `actual` variable does not contain any NA values. You need to define a variable called `budget`, which requires much less detail than `actual`. For example, `budget` only needs 10 percent of the `line` dimension values, while `actual` needs all of them.

If you define `budget` without setting sparsity, then all of the `line` dimension values are present for every `month` and `division`, but 90 percent of the `line` dimension cells will have NA values.

To handle sparse data in this case, you define `budget` with an unnamed composite for only the `line` dimension as shown below.

```
DEFINE budget DECIMAL <SPARSE <line> division month>
```

Defining Hierarchical Dimensions and Variables That Use Them

A hierarchical dimension is a means of organizing and structuring parent-child (one-to-many) data within a single dimension and using self-relations to organize the values of the hierarchical dimension into groups. A hierarchy exists when values within a dimension are arranged in levels, with each level representing the aggregated total of the data from the level below. Some dimensions have multiple hierarchies.

Hierarchical dimensions allow you to store data of varying levels of aggregation within a single variable. This type of storage affords a quicker response time for users who want to view the data, particularly when the variable is large.

Rather than defining two separate dimensions, one for city and the other for region, you could define a hierarchical dimension named `geography` that contains both city and region values.

```
GEOGRAPHY
--------------
EAST
WEST
BOSTON
SAN FRANCISCO
SEATTLE
```

Defining a Variable with a Hierarchical Dimension

You use a hierarchical dimension to define a variable that contains data of varying levels of aggregation within a single variable. This type of storage affords a quicker response time for users who want to view the data, particularly when the variable is large.

Frequently, the cells in the variable that correspond to upper level values in the hierarchical dimension contain the sum or total of the values in the cells of the variable that correspond to the lower level dimension values. For example, in a `sales` variable that is defined with a hierarchical dimension representing time, the cells of the variable for each quarter might represent the total sales for the months in the quarter.

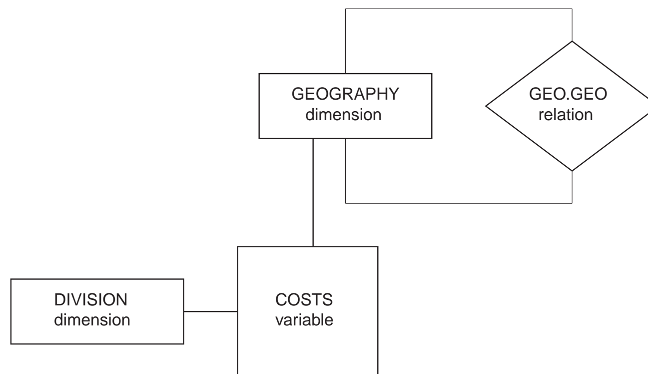
After you have defined a variable with hierarchical dimensions, you can add variable data to the lowest level of the hierarchy, and then calculate or **aggregate** the values for the higher levels of the hierarchy. Conversely, you can distribute or **allocate** data from higher levels to lower levels of the hierarchy.

See Also:

- Chapter 9, "Allocating Data" for information about allocating data.
- Chapter 12, "Aggregating Data" for information about aggregating data.

Example: Variable with a Hierarchical Dimension

The conceptual diagram below illustrates the `geography` dimension that contains values for both cities and regions, the `geo.geo` relation that defines the relationships between cities and regions, the `division` dimension that contains the list of divisions, and the `costs` variable that contains the expenses for each `division` by city and the totals by region.



The `division` and `geography` dimensions have the following values.

```
DIVISION
--------------
DIVA
DIVB

GEOGRAPHY
--------------
EAST
WEST
BOSTON
SAN FRANCISCO
SEATTLE
```

Assume that the `geo.geo` relation was defined using the following command.

```
define geo.geo relation geography <geography>
```

After region values have been assigned to the city values in the `geo.geo` self-relation, a report of `geo.geo` produces the following.

```
GEOGRAPHY       GEO.GEO
-------------- ----------
EAST           NA
WEST           NA
BOSTON         EAST
SAN FRANCISCO  WEST
SEATTLE        WEST
```

If you enter data at the lowest level (city level) of `costs`, then it has the values shown below.

```
          ------------------------COSTS------------------------------
          ----------------------GEOGRAPHY----------------------------

DIVISION     EAST       WEST       BOSTON   SAN FRANCISCO      SEATTLE
--------- ---------- ---------- ---------- -------------    ----------
DIVA           NA      NA       27,600.00  10,000.00       40,000.00
DIVB           NA      NA       30,000.00  12,000.00       50,000.00
```

After you aggregate the data, the `costs` variable has values in all of its cells, including the cells for the totals for the East and West regions.

```
          ------------------------COSTS------------------------------
          ----------------------GEOGRAPHY----------------------------

DIVISION     EAST       WEST       BOSTON   SAN FRANCISCO      SEATTLE
--------- ---------- ---------- ---------- -------------    ----------
DIVA       27,600.00  50,000.00  27,600.00  10,000.00        40,000.00
DIVB       30,000.00  62,000.00  30,000.00  12,000.00        50,000.00
```

Defining Concat Dimensions and Variables That Use Them

A concat dimension combines two or more base dimensions into a single dimension. You can use a concat dimension instead of a hierarchical simple dimension as another means of organizing and structuring parent-child data within a dimension. You use self-relations to organize the values of the concat dimension into groups by the levels of the hierarchy.

In a relational dimension table, suppose you have one column for districts with city names as its values, and another column for regions. You can define a `district` dimension and a `region` dimension in your analytic workspace and load into them the values of the relational columns. Those dimensions have the following values.

```
DISTRICT
----------
BOSTON
SAN FRANCISCO
SEATTLE

REGION
-------
EAST
WEST
```

You can define a concat dimension named `reg.dist.ccdim` based on those simple flat dimensions. The concat dimension contains the values of both dimensions.

```
REG.DIST.CCDIM
-------------------------
<REGION: EAST>
<REGION: WEST>
<DISTRICT: BOSTON>
<DISTRICT: SAN FRANCISCO>
<DISTRICT: SEATTLE>
```

You can then define a self-relation that groups the values of the concat dimension into hierarchical levels. As with a hierarchical simple dimension, you can use a concat dimension to define a variable that contains different levels of aggregation.

Example: Variable with a Concat Dimension

You can define a variable dimensioned by the `reg.dist.ccdim` concat dimension and the `division` dimension from "Example: Variable with a Hierarchical Dimension" on page 3-24 as follows.

```
DEFINE costs VARIABLE DECIMAL <reg.dist.ccdim division>
```

You can define a self-relation for the `reg.dist.ccdim` concat dimension that identifies the parent-to-child relationships of the district-region hierarchy as follows.

```
DEFINE rdccdim.rdccdim RELATION reg.dist.ccdim <reg.dist.ccdim>
limit district to 'BOSTON'
rdccdim.rdccdim(REG.DIST.CCDIM district) = reg.dist.ccdim(REGION 'EAST')
limit district to 'DENVER' 'SEATTLE'
rdccdim.rdccdim(REG.DIST.CCDIM district) = reg.dist.ccdim(REGION 'WEST')
```

If you enter data at the lowest level (the `district` dimension level), then the `costs` variable has the values shown below.

```
          ------------------------COSTS------------------------------
          --------------------REG.DIST.CCDIM-------------------------
           <REGION:   <REGION:  <DISTRICT: <DISTRICT:       <DISTRICT:
DIVISION    EAST>      WEST>     BOSTON>    SAN FRANCISCO>  SEATTLE>
--------- ---------- ---------- ---------- ---------------- ----------
DIVA         NA         NA         27,600.00  10,000.00       40,000.00
DIVB         NA         NA         30,000.00  12,000.00       50,000.00
```

You can aggregate the data in the `costs` variable by creating an aggregation map and then using the `AGGREGATE` command. After you aggregate the data, the `costs` variable has values in all of its cells, including the cells for the totals for the `EAST` and `WEST` regions.

```
          ------------------------COSTS------------------------------
          --------------------REG.DIST.CCDIM-------------------------

           <REGION:   <REGION:  <DISTRICT: <DISTRICT:       <DISTRICT:
DIVISION    EAST>      WEST>     BOSTON>    SAN FRANCISCO>  SEATTLE>
--------- ---------- ---------- ---------- ---------------- ----------
DIVA       27,600.00  50,000.00  27,600.00  10,000.00       40,000.00
DIVB       30,000.00  62,000.00  30,000.00  12,000.00       50,000.00
```

Changing the Definition of an Object

The definition of the last object you have defined in your analytic workspace is the current definition. You can append characteristics, such as a description, property, or permission to the current definition. If you want to append a characteristic to a definition that is not current, then you can use the `CONSIDER` command to make it the current definition.

The following table lists the OLAP DML commands that you can use to append characteristics to an object definition.

Command	Description
`AGGMAP`	Allows you to specify completely new contents for a new or existing `AGGMAP` type aggregation map, which you can use with the `AGGREGATE` command
`ALLOCMAP`	Allows you to specify completely new contents for a new or existing `ALLOCMAP` type aggregation map, which you can use with the `ALLOCATE` command

Command	Description
EQ	Allows you to specify the expression to be calculated for a formula that has already been defined
LD	Assigns a long description to an object definition
MODEL	Allows you to specify completely new contents for a new or existing model
PERMIT	Assigns access permission to an object definition
PROGRAM	Allows you to specify completely new contents for a new or existing program
PROPERTY	Assigns a property to an object definition

Suppose that you have defined a Boolean variable named `onplan`. Later, you want to add a description to the definition of the variable.

As shown below, to change the definition of the `ONPLAN` variable, you first make `ONPLAN` the current definition, and then you append a description to the definition.

```
CONSIDER onplan
LD Are these districts tracked on a special plan?
```

You can redefine some characteristics of a variable definition by using the `CHGDFN` command. In the following example, the segment size of the `sales` variable is changed.

```
CHGDFN sales SEGWIDTH 150 1000
```

For more information on the `DEFINE` and `CHGDFN` commands, see the topics for these commands in the Oracle9i OLAP DML Reference help.

4 Working with Expressions

Expressions represent data values in the syntax of the OLAP DML. This chapter explains how to create and use expressions. It includes the following topics:

- Introducing Expressions
- Dimensionality of Expressions
- Specifying a Single Value for the Dimension of an Expression
- Using Workspace Objects in Expressions
- Numeric Expressions
- Text Expressions
- Boolean Expressions
- Conditional Expressions
- Substitution Expressions
- Working with NA Values

Introducing Expressions

Expressions represent data values in the syntax of the OLAP DML. You can use expressions as arguments in commands or functions and as values for options. An expression often performs a mathematical or logical operation. It always evaluates to a result in one of the workspace data types.

An expression can be:

- A single, literal value (for example, `10` or `'EAST'`)
- A variable or formula that contains multiple values (for example, `sales`)
- A function that returns one or more values (for example, `TOTAL` or `JOINLINES`)
- A calculation that combines literal values, dimensions, variables, formulas, and functions with operators (for example, `inflation*1.02` or `actual gt 20000`)

An expression has a data type. It can also have dimensions. The data type and dimensions of an expression depend on the values you are using in the expression.

Data Types of Expressions

The data type of an expression can be one of the following basic types:

- Numeric
- Text
- Date (evaluating to a date value)
- Boolean (evaluating to a `YES` or `NO` value)

These data types are defined in "Data Types" on page 3-4.

How the Data Type of an Expression is Determined

The data type of an expression is the data type of the resulting value. It may not be the same as the data type of the data objects that make up the expression; it depends on the data and on the operators and functions that are involved.

In addition, a conditional expression that is indicated by an `IF... THEN. . . ELSE` operator is supported. A conditional expression returns a value whose data type depends on the expressions in the `THEN` and `ELSE` clauses, not on the expression in the `IF` clause, which must be Boolean.

Note: Do not confuse a conditional expression with the IF command, which has similar syntax but a different purpose. The IF command does not have a data type and is not evaluated like an expression.

Changing the Data Type of an Expression

You can use the CONVERT function to change data type of an expression. For example, you can convert a number to text, or you can convert a text string that consists of digits to a number.

However, there is no need to convert data to another type within the same basic category because those conversions are made automatically. In general, you can use TEXT, NTEXT, or ID data anywhere text is called for, and you can use integers and decimal numbers interchangeably.

OLAP DML data types are discussed in "Data Types" on page 3-4.

Operators

An operator is a symbol that transforms a value or combines it in some way with another value. The following table describes the categories of OLAP DML operators.

Table 4–1 OLAP DML Operators

Category	Description
Arithmetic	Operators that you can use in numeric expressions with numeric data to produce a numeric result. You can also use some arithmetic operators in date expressions with a mix of date and numeric data, which returns either a date or numeric result. For more information on arithmetic operators, see Table 4–2, "Arithmetic Operators".
Assignment	An operator that you use to create an assignment statement that stores the results of an expression into an object. For more information on using assignment statements, see "Using Objects in Assignment Statements" on page 5-11.
Comparison	Operators that you can use to compare two values of the same basic type (numeric, text, date, or, in rare cases, Boolean), which returns a Boolean result. For more information on comparison operators, see Table 4–3, "Boolean Operators".

Table 4–1 OLAP DML Operators

Category	Description
Conditional	Operators that you can use to select one of two values based on a Boolean condition. For more information on the substitution operator, see "Conditional Expressions" on page 4-29.
Logical	Operators that you can use to transform Boolean values using logical operations, which returns a Boolean result. For more information on logical operators, see Table 4–3, "Boolean Operators".
Substitution	An operator that you can use to evaluate an expression and substitute the resulting value. For more information on the substitution operator, see "Substitution Expressions" on page 4-30.

Saving an Expression

You can save an expression in a formula. Typically, you define a formula to save complex or frequently used expressions. A formula is an object that you name and define using the `DEFINE FORMULA` command.

For example, you can define a formula to calculate dollar sales, as follows.

```
DEFINE dollar.sales FORMULA units * price
```

Each time you use a formula, the expression it represents is evaluated. The results are not stored.

Dimensionality of Expressions

An expression is dimensioned by a union of the dimensions of all the variables, dimensions, relations, formulas, qualified data references, and functions in the expression.

Item	Dimensioned By	Comments
Variable Relation Formula	The dimensions listed in the definition of the object	**Example 1:** If the `price` variable is dimensioned by `month` and `product`, then the expression `price * 1.2` is also dimensioned by `month` and `product`. **Example 2:** If the `units` variable is dimensioned by `month`, `product`, and `district`, then the expression `units * price` is dimensioned by `month`, `product`, and `district` (even though the dimensions of the `price` variable are `month` and `product` only).
Qualified data reference	All of the dimensions of the associated object, except for the dimensions being qualified	Qualified data references are described in "Specifying a Single Value for the Dimension of an Expression" on page 4-6.
Function	In most cases, the union of the dimensions of its input arguments	Unless otherwise noted in the OLAP DML Reference, when you specify breakout dimensions or relations in an aggregation function, you change the dimensionality of the expression. The first dimension that you specify as a breakout dimension is the slowest varying and the last dimension that you specify is the fastest varying.

Determining the Dimensions of an Expression

You can find out the dimensions of an expression with the `PARSE` command and the `INFO` function. `PARSE` evaluates the text of an expression; the `INFO` function indicates how the expression is interpreted.

This example illustrates the use of the `DIMENSION` keyword with the `INFO` function to retrieve the dimensions of the expression just analyzed by the `PARSE` command. The following commands produce the output shown below them.

```
PARSE 'TOTAL(sales region)'

SHOW INFO(PARSE DIMENSION)
REGION
```

How Dimension Status Affects the Results of Expressions

The number of values an expression yields depends on the dimensions of the expression and the status of those dimensions. An expression yields one data value for each combination of dimension values in the current status. For example, if three dimension values are in status for `month`, and two for `product`, then the expression `price gt 100` results in six values (3 times 2).

Thus, to get the desired results, you must ensure that the dimensions of an expression are limited to the range of data you want to consider. In addition, you must take into consideration any `PERMIT` commands that might limit access to the dimensions of the data.

> **See Also:** Chapter 6, "Selecting Data" for more information about setting the status of a dimension.

Specifying a Single Value for the Dimension of an Expression

A qualified data reference (QDR) is a way of limiting one or more dimensions of an expression to a single value. QDRs are useful when you want to specify a single value without changing the current status. Using a QDR, you can qualify a dimension (which allows you to specify one dimension value in an expression) or one or more dimensions of a variable or relation.

A qualified data reference takes the following form.

```
expression(dimname1 dimexp1 [, dimname2 dimexp2. . .])
```

The *`dimname`* argument is the name of one of the dimensions, or a dimension surrogate of the dimension, of the expression and the *`dimexp`* argument is one of the following:

- A value of *`dimname`*.

- A text expression whose result is a value of *dimname*.
- A numeric expression whose result is the logical position of a value of *dimname*.
- A relation of *dimname*.

Note: To qualify a complex expression, use the QUAL function.

Qualifying a Variable

You can qualify any or all of a dimensions of a variable using either of the following techniques:

- The QDR can temporarily limit a dimension of the variable by selecting one specified value of the dimension. This value can be outside the current status.
- The QDR can replace a dimension of the variable with a less aggregate related dimension when you supply the name of an appropriate relation as the qualifier. The dimension is temporarily replaced by the dimension(s) of the relation.

For example, the variable sales has three dimensions, month, product, and district. You might want to compare total sales in Boston to the total sales in all cities. In a single command, you want district to be limited to *two* different values:

- For the numerator of the expression, you want the status of district to be BOSTON.
- For the denominator of the expression, you want the status of district to be ALL.

The command below lets you calculate this result by using a QDR.

```
SHOW sales(district 'BOSTON')/TOTAL(sales)
```

You can qualify more than one of the dimensions of a variable. For example, if you qualify all the dimensions of the sales variable by specifying one dimension value of each dimension, then you narrow sales down to a single–cell value.

To fetch sales for JUN02, TENTS, and SEATTLE, use the following QDR.

```
SHOW sales(month 'JUN02', product 'TENTS', district 'SEATTLE')
```

This command fetches a single value.

You can use a qualified data reference with the target expression of the = command. This lets you assign a value to a specific cell in a data object.

The following example assigns the value 10200 to the data cell of the `sales` composite that is specified in the qualified data reference. If the composite named `sales` does not already have a value for the combination `BOSTON` and `TENTS`, then this value combination is added to the composite, thus adding the data cell.

```
sales(market 'BOSTON' product 'TENTS' month 'JAN99')= 10200
```

Replacing a Dimension in a Variable

When you use a relation as the qualifier in the QDR, you replace a dimension of the variable with the dimension or dimensions of the relation. The relation must be related to the dimension that you are qualifying, and it must be dimensioned by the replacement dimension.

Example 4–1 Replacing a Dimension in a Variable

Suppose you have two variables, `sales` and `quota`, which are dimensioned by `month`, `product`, and `district`. A third variable, `division.mgr`, is dimensioned by `month` and `division`. You also have a relation between `division` and `product`, called `division.product`. These objects have the following definitions.

```
DEFINE SALES VARIABLE DECIMAL <MONTH PRODUCT DISTRICT>
LD Sales Revenue
DEFINE QUOTA VARIABLE DECIMAL <MONTH PRODUCT DISTRICT>
DEFINE DIVISION.MGR VARIABLE TEXT <MONTH DIVISION>
DEFINE DIVISION.PRODUCT RELATION DIVISION <PRODUCT>
LD DIVISION for each PRODUCT
```

The command below produces the report following it.

```
REPORT division.mgr

-------------------DIVISION.MGR----------------------
          ----------------------MONTH--------------------------
DIVISION  JAN02    FEB02    MAR02    APR02    MAY02    JUN02
--------  -------- -------- -------- -------- -------- --------
CAMPING   Hawley   Hawley   Jones    Jones    Jones    Jones
SPORTING  Carey    Carey    Carey    Carey    Carey    Musgrave
CLOTHING  Musgrave Musgrave Musgrave Musgrave Musgrave Wong
```

Suppose you want to obtain a report that shows the fraction by which sales have exceeded quota; and you want to include the appropriate division manager for each product. You can show the division manager for each product by using the relation `division.product`, which is related to `division` and dimensioned by `product`, as the qualifier. The QDR replaces the `division` dimension with `product`, so that it has the same dimensions as the other expression in the report "`sales / quota`." The command below produces the report following it.

```
REPORT DOWN month sales W 6 sales/quota W 8 HEADING -
   'MANAGER' division.mgr(division division.product)

DISTRICT: BOSTON
        -----------------------------PRODUCT-----------------------------------
        ----TENTS---- ---CANOES---- --RACQUETS---  --SPORTSWEAR-- ---FOOTWEAR---
        SALES/        SALES/        SALES/         SALES/         SALES/
MONTH   QUOTA MANAGER QUOTA MANAGER QUOTA MANAGER  QUOTA MANAGER  QUOTA  MANAGER
------  ----- ------- ----- ------- ----- -------  ----- -------- ----- --------
JAN02   1.00  Hawley  0.82  Hawley  1.02  Carey    0.91  Musgrave 0.92  Musgrave
FEB02   0.84  Hawley  0.96  Hawley  1.00  Carey    0.80  Musgrave 1.07  Musgrave
MAR02   0.87  Jones   0.95  Jones   0.87  Carey    0.88  Musgrave 0.91  Musgrave
APR02   0.91  Jones   0.93  Jones   0.99  Carey    0.94  Musgrave 0.95  Musgrave
.
.
.
```

Qualifying a Relation

You can also use a QDR to qualify a relation (which is really a special kind of variable).

Suppose the `region.district` relation is dimensioned by `district`. If you qualify `district` with the value `SEATTLE`, then the value of the expression is the value of the relation for `SEATTLE`. Because the QDR specifies one value of `district`, the expression has a single–cell result.

The definition of `region.district` is as follows.

```
DEFINE REGION.DISTRICT RELATION REGION <DISTRICT>
LD The region for each district
```

The command below displays the value `WEST`.

```
SHOW region.district(district 'SEATTLE')
```

Qualifying a Dimension

You can use a QDR to qualify the dimension itself, which allows you to specify one dimension value in an expression. The following expression specifies one value of `district`, the one contained in the single-cell variable `mydistrict`.

```
district(district mydistrict)
```

For a concat dimension, you can use a QDR to qualify the dimension by specifying a value from one of the base dimensions of the concat dimension. The following expression specifies one value of `reg.dist.ccdim`, a concat dimension that has `region` and `district` as its base dimensions. The costs variable is dimensioned by the `division` and `reg.dist.ccdim` dimensions.

```
show reg.dist.ccdim(district 'BOSTON')
```

The preceding expression produces the following result.

```
<DISTRICT: BOSTON>
```

Using Ampersand Substitution with QDRs

An ampersand character (&) at the beginning of an expression substitutes the value of the expression for the expression itself in a command or function.When you use an ampersand with a QDR, you must enclose the whole expression in parentheses if you want the variable to be qualified before the substitution is made.

Suppose you have a text variable named `myvar` that is dimensioned by `reptype` and that contains the names of variables. Remember that it is `myvar` that is dimensioned by `reptype`, not the variables named by `myvar`. Therefore, you must use parentheses so that `myvar` is qualified and the resulting value is used in the `REPORT` command.

```
REPORT &(myvar(reptype 'ACTUAL'))
```

If you do not use parentheses and the variable that is specified in `myvar` is `sales`, then you will get an error message that `sales` is not dimensioned by `reptype`.

Using the QUAL Function to Specify a QDR

Sometimes you will find that the syntax of a QDR is ambiguous and could either be misinterpreted or cause a syntax error. In this case, you can use the `QUAL` function to explicitly specify a qualified data reference (QDR).

Example 4–2 Using the QUAL Function

The following example first shows how you might view your data by limiting its dimensions, and then how you might view it by using `QUAL`.

These commands produce the report shown below them.

```
LIMIT month TO 'JAN96' TO 'JUN96'
LIMIT line TO 'COGS'
LIMIT division TO 'SPORTING'
REPORT DOWN month W 11 MAX(actual,budget) W 11 actual W 11 budget

DIVISION: SPORTING
               ---------------LINE----------------
               ---------------COGS----------------
               MAX(ACTUAL,
MONTH            BUDGET)     ACTUAL      BUDGET
-------------- ----------- ----------- -----------
JAN96           287,557.87  287,557.87  279,773.01
FEB96           323,981.56  315,298.82  323,981.56
MAR96           326,184.87  326,184.87  302,177.88
APR96           394,544.27  394,544.27  386,100.82
MAY96           449,862.25  449,862.25  433,997.89
JUN96           457,347.55  457,347.55  448,042.45
```

Now consider how you might view the same figures for `MAX(actual,budget)` without changing the status of `line` or `division`.

```
ALLSTAT
LIMIT month TO 'JAN96' TO 'JUN96'
REPORT HEADING 'For Cogs in Sporting Division' DOWN month -
   W 11 HEADING 'MAX(actual,budget)'-
   QUAL(MAX(actual,budget), line 'COGS', division 'SPORTING')

For Cogs in
Sporting       MAX(ACTUAL,
Division         BUDGET)
-------------- -----------
JAN96           287,557.87
FEB96           323,981.56
MAR96           326,184.87
APR96           394,544.27
MAY96           449,862.25
JUN96           457,347.55
```

If you attempt to produce the same report with standard QDR syntax, then an error is signalled.

```
REPORT HEADING 'For Cogs in Sporting Division' DOWN month -
   W 11 HEADING 'MAX(actual,budget)'-
   MAX(actual,budget) (line cogs, division sporting)
```

The following error message is produced.

```
ERROR: A right parenthesis or an operator is expected after LINE.
```

Using Workspace Objects in Expressions

You can use objects in expressions as described below:

- You can use a dimension, a dimension surrogate, a relation, or a variable as an array of data by specifying the name of the object.
- You can use a formula or a function as a sub-expression or as an expression in a command or function by specifying the name of the formula or the function.
- You can use a valueset as a list of dimension values in an expression by specifying the name of the valueset.
- You can use various data objects as the target or source expression in an assignment statement.

Using Dimensions or Dimension Surrogates in Expressions

In expressions, a dimension or dimension surrogate is referenced as a one-dimensional array.

If the dimension or surrogate has a data type of `TEXT`, then, in most cases, its values are referenced as text values. `NUMBER` dimension values are always referenced by the value itself.

However, for dimension types other than `NUMBER` dimension values are referenced by their numeric positions in the dimension array when you do one of the following:

- Use a dimension with a data type of `TEXT` in a numeric expression
- Compare one value in a dimension to another value in the same dimension

In these cases, the integer position number is based on the default status list, not on the current status.

Using Composites in Expressions

In expressions, composites behave much the same way that dimensions do and, generally, you can use a composite in an expression anywhere you can use a dimension:

- If the composite is named, then specify its name.
- If the composite is unnamed, then specify `SPARSE <dimensions...>`.

Using Variables in Expressions

In expressions, a variable is referenced as an array containing values of the specified data type.

When you assign values to a variable or when you use `REPORT` or another command or function that loops over the dimensions of a variable, the values of the fastest-varying dimension of the variable vary first. For example, for the `opcosts` variable that is dimensioned by `month` and `city`, when you view the variable as `REPORT` command output, you see the data for all months for the first city before you see any data for the second city. In this case, `month` is the fastest-varying dimension because its values change before those of `city`. When you write programs that loop over a multidimensional variable in this way, try to maximize performance by matching the fastest-varying dimension with the inner loop.

Note: When you use a variable as the solution variable in a model, the model will execute most efficiently if the order of the dimensions in the definition of the solution variable matches the order of the dimensions in the `DIMENSION` commands in the model.

You can uniquely and completely select any item of data within a multidimensional variable by using a qualified data reference (QDR) to specify one value from each of the dimensions of the variable.

For example, if the `opcosts` variable is dimensioned by `month` and `city`, specifying `'JAN02'` for the `month` dimension and `'BOSTON'` for the `city` dimension uniquely specifies a single cell in the variable.

Using Variables Defined with Composites in Expressions

In most cases, when you use functions and commands with variables that are defined with composites, the functions and commands treat those variables as if they were defined with base dimensions:

- You can access a variable that is dimensioned by a composite by requesting any of the base dimension values.
- The values of a composite that are **in status** are determined by the status of the base dimensions of the composite. Composites are not dimensions, and therefore, they do not have any independent status.

Default Behavior of Commands That Loop Over Variables

When you use the REPORT command or any other command that loops over a variable that uses a composite, the default behavior is to evaluate all the combinations of the values of the base dimensions of the composite that are in status. Any combinations that do not exist in the composite display NA for their associated data.

For example, the following commands create a report for the East region that shows the number of coupons issued for sportswear from January through March 2002. Since no coupons were issued in March 2002, the report displays NA in that column.

```
LIMIT month TO 'JAN02' 'FEB02' 'MAR02'
LIMIT market TO 'EAST'
LIMIT product TO 'SPORTSWEAR'
REPORT coupons

MARKET: EAST
               ------------COUPONS-------------
               -------------MONTH--------------
PRODUCT          JAN02      FEB02      MAR02
-------------- ---------- ---------- ----------
SPORTSWEAR          1,000      1,000         NA
```

However, for performance reasons, you can change the default looping behavior for commands such as REPORT, ROW, and = so that they loop over the values in the composite rather than all of the base dimension values.

Using Relations In Expressions

A relation is, in many ways, just a special type of variable. Instead of holding general data values, a relation contains values of the related dimension. Consequently, in an expression, a relation behaves somewhat like a variable and somewhat like a dimension:

- When you use a relation in a text expression, the relation value is referenced as a text value. The values of the related dimension that is contained in the relation are converted into text, and you can use these values in an expression. You can also compare a text literal to a relation.
- When you use a relation in a numeric expression, the relation value is referenced by its position (an integer) in its related dimension array. You can use this numeric value in an expression. The position number is based on the default status list of the dimension, not the current status list of the dimension.

Using Functions in Expressions

A function is a predefined calculation that returns a value. A number of built-in functions are provided, including:

- Numeric functions. You can use these functions to make calculations and analyze data.
- Date functions. You can use these functions to manipulate dates.
- Text functions. You can use these functions to join characters or lines, search for or extract a group of characters, or calculate the length of the text.

Numeric Expressions

A numeric expression evaluates to data with any of the numeric data types (that is, `INTEGER`, `SHORTINTEGER`, `DECIMAL`, `SHORTDECIMAL`, and `NUMBER`). The data in a numeric expression can be any combination of the following:

- Numeric literals
- Numeric variables or formulas
- Dimensions
- Functions that yield numeric results
- Date literals, variables, formulas, or functions

In addition, you can join any of these three-part expressions with the arithmetic operators for a more complex numeric expression. You use arithmetic operators in numeric expressions with numeric data, which returns a numeric result. You can also use some arithmetic operators in date expressions with a mix of date and numeric data, to retrieve either a date or numeric result.

Arithmetic Operators

The following table shows the OLAP DML arithmetic operators. When you use two or more operators in a numeric expression, the expression is evaluated according to standard rules of arithmetic. The column entitled Priority indicates the order in which that operator is evaluated. Operators of the same priority are evaluated from left to right.

Table 4–2 Arithmetic Operators

Operator	Operation	Priority
-	Sign reversal	1
**	Exponentiation	2
* and /	Multiplication and division	3
+ and -	Addition and subtraction	4

Note: A comma is required before a negative number that follows another numeric expression, or the minus sign is interpreted as a subtraction operator. For example, `intvar,-4`.

Mixing Numeric Data Types

You can include any type of numeric data in the same numeric expression.

The data type of the result is determined according to the following rules.

IF . . .	THEN the result is . . .
all the data in the expression is INTEGER or SHORTINTEGER, and the only operations are addition, subtraction, and multiplication,	INTEGER.
any of the data is NUMBER,	NUMBER.
any of the data is DECIMAL or SHORTDECIMAL, and no data is NUMBER,	DECIMAL.
you perform any division or exponentiation operations,	DECIMAL.

Automatic Conversion of Numeric Data Types

Numbers are converted to different data types according to the following rules.

IF you . . .	THEN . . .
use a value with the SHORTINTEGER or SHORTDECIMAL data type in an expression,	the value is converted to its long counterpart before using it. **Note:** See "Boolean Expressions" on page 4-21 for information about problems that can occur when you mix SHORTDECIMAL and DECIMAL data types in a comparison expression.
save the results of a calculation as a value with the SHORTINTEGER data type,	NA is stored when the result is outside the range of a SHORTINTEGER (-32768 to 32767).
assign the value of a DECIMAL expression to an object with the INTEGER data type,	the value is rounded before storing or using it. **Note:** If the decimal value is outside the range of an integer (approximately plus or minus 2 billion), then an NA is stored.
use a decimal value where a value with the INTEGER data type is required,	the value is rounded before storing or using it. **Note:** If the decimal value is outside the range of an integer (approximately plus or minus 2 billion), then an NA is stored.

IF you . . .	THEN . . .
assign the value of a decimal expression to a variable with the SHORTDECIMAL data type,	only the first 7 significant digits are stored.
combine NUMBER values with other numeric data types,	all values are converted to NUMBER.

If these conversions are not what you want, then you can use the CONVERT, TO_CHAR, TO_NCHAR, TO_NUMBER, or TO_DATE functions to get different results.

Using Dimensions in Arithmetic Expressions

When you use a dimension with a data type of TEXT in a numeric expression, the dimension value is treated as a position (an integer) and is used numerically. The position number is based on the default status list, not on current status.

Using Dates in Arithmetic Expressions

When you use dates in arithmetic expressions, the result can be numeric or it can be a date. The following table shows the legal operations for dates and the data type of the result.

IF you.... . . .	THEN the result is... . . .
add or subtract a number from a date,	a future or prior date.
subtract a date from a date,	the number of days between them.
add a date to a date,	an error; this is an invalid operation.
add or subtract a number from a time period,	the time period at the appropriate interval in the future or the past, similar to the return values of the LEAD or LAG function. The result is NA when there is no dimension value that corresponds to the result. The calculation is made based on the positions of the values in the default status list of the dimension.

Limitations of Floating Point Calculations

All decimal data are converted to floating point format, both for storing and for calculations. In floating point format, a number is represented by means of a mantissa and an exponent. The mantissa and the exponent are stored as binary

numbers. The mantissa is a binary fraction which, when multiplied by a number equal to 2 raised to the exponent, produces a number that equals or closely approximates the original decimal number.

Because there is not always an exact binary representation for a fractional decimal number, just as there is not an exact representation for the decimal value of 1/3, fractional parts of decimal numbers cannot always be represented exactly as binary fractions. Arithmetic operations on floating point numbers may result in further approximations, and the inaccuracy will gradually increase with the number of operations. In addition to the approximation factor, the available number of significant digits affects the exactness of the result.

For all of these reasons, a result computed by the `TOTAL`, `AVERAGE`, or other aggregation functions on a `DECIMAL` or `SHORTDECIMAL` variable may differ in the least significant digits from a result you compute by hand. Because the `SHORTDECIMAL` data type provides a maximum of only seven significant digits, you will see more of these differences with `SHORTDECIMAL` data. Therefore, you might want to use the `NUMBER` data type when accuracy is more important than computational speed, such as variables that contain currency amounts.

Another result of the fact that some fractional decimal numbers cannot be exactly represented by binary fractions is that for such numbers, the `DECIMAL` data type will offer a different and closer approximation than the `SHORTDECIMAL` data type, because it has more significant digits. This can lead to problems when `SHORTDECIMAL` and `DECIMAL` data types are mixed in a comparison expression. See the topic "Boolean Expressions" on page 4-21 for information on how to handle such comparisons.

Controlling Errors During Calculations

You can control the following types of errors:

- Division by zero. If you divide an NA value by zero, then the result is NA; no error occurs. Dividing a non-NA value by zero normally produces an error. If a divide-by-zero error occurs when you are making a calculation on dimensioned data, then you can end up with partial results. When you use the `REPORT` or the = command, values are reported or stored as they are calculated, so the division by zero halts the loop before it has gone through all the values.

  If you want to suppress the divide-by-zero error, then you can change the value of the `DIVIDEBYZERO` option to `YES`. This means that the result of any division by zero is NA and no error occurs. This allows the calculation of the other values of a dimensioned expression to continue.

- Root of negative numbers. It is normally an error to try to take the root of a negative number (which includes raising a number to a non-integer power). If you want to suppress the error message and allow the calculation of roots for non-negative values of the expression to continue, then set the ROOTOFNEGATIVE option to YES.
- Overflow errors. The DECIMALOVERFLOW option works in a similar manner to DIVIDEBYZERO. It lets you control whether an error is generated when a calculation produces a decimal result larger than it can handle.

Text Expressions

A text expression evaluates to data with the TEXT, NTEXT, or ID data type. Text expressions can be any combination of the following:

- Text literals; for example, 'BOSTON' or 'Current Sales Report'
- Text dimensions; for example, district or month
- Text variables or formulas; for example, product.name
- Functions that yield text results; for example, JOINLINES('Product: ' product.name)

Suppose textvar is a variable whose value is 'geog', which is the name of a dimension. Whether you enclose the word textvar in quotation marks determines whether the following OBJ function calls return the word VARIABLE (the type of object textvar is) or DIMENSION (the type of object geog is).

```
SHOW OBJ(TYPE 'textvar')
VARIABLE

SHOW OBJ(TYPE textvar)
DIMENSION
```

Working with Dates in Text Expressions

If you use a DATETIME value where a text value (TEXT, NTEXT, or ID) is expected, or if you store a DATETIME value in a text variable, then the DATETIME value is automatically converted to a text value.

The format of a DATETIME value is controlled by the NLS_DATE_FORMAT option. Once a DATETIME value is stored in a text variable, the NLS_DATE_FORMAT setting has no impact.

Working with NTEXT Data

TEXT and NTEXT data are interchangeable in most cases. However, implicit conversion can occur, such as when an NTEXT value is assigned to a TEXT variable. When TEXT is converted to NTEXT, no data loss occurs because the UTF-8 character encoding of the NTEXT data type encompasses most other data types. However, when NTEXT is converted to TEXT, data loss will occur if NTEXT characters are not represented in the workspace character set.

When TEXT and NTEXT values are used together, for example in a call to the JOINCHARS function, the TEXT value is converted to NTEXT and an NTEXT value is returned.

Boolean Expressions

A Boolean expression is a logical statement that is either true or false. Boolean expressions can compare data of any type as long as both parts of the expression have the same basic data type. You can test data to see if it is equal to, greater than, or less than other data.

A Boolean expression can consist of Boolean data, such as the following:

- Boolean values (YES and NO, and their synonyms, ON and OFF, and TRUE and FALSE)
- Boolean variables or formulas
- Functions that yield Boolean results
- Boolean values calculated by comparison operators

For example, if you have the Boolean expression shown below, then each value of the variable actual is compared to the constant 20,000. If the value is greater than 20,000, then the statement is true; if the value is less than or equal to 20,000, then the statement is false.

```
actual GT 20000
```

When you are supplying a Boolean value, you can type either yes, on, or true for a true value, and no, off, or false for a false value. When the result of a Boolean calculation is produced, the defaults are yes and no in the language specified by the NLS_LANGUAGE option. The read-only YESSPELL and NOSPELL options record the yes and no values.

The following table shows the comparison operators and the logical operators. You use these operators to make expressions in much the same way as arithmetic

operators. The column entitled “Priority” indicates the order in which that operator is evaluated.

Table 4–3 Boolean Operators

Operator	Operation	Example	Priority
NOT	Returns opposite of Boolean expression	NOT(yes) = no	1
EQ	Equal to	4 EQ 4 = yes	2
NE	Not equal to	5 NE 2 = yes	2
GT	Greater than	5 GT 7 = no	2
LT	Less than	5 LT 7 = yes	2
GE	Greater than or equal to	8 GE 8 = yes	2
LE	Less than or equal to	8 LE 9 = yes	2
IN	Is a date in a time period?	'1JAN02' IN W1.02 = yes	2
LIKE	Does a text value match a specified text pattern?	'FINANCE' LIKE '%NAN%' = yes	2
AND	Both expressions are true	8 GE 8 AND 5 LT 7 = yes	3
OR	Either expression is true	8 GE 8 OR 5 GT 7 = yes	4

Each operator has a priority that determines its order of evaluation. Operators of equal priority are evaluated left to right, unless parentheses change the order of evaluation. However, the evaluation is halted when the truth value is already decided. For example, in the following expression, the TOTAL function is never executed because the first phrase determines that the whole expression is true.

```
yes EQ yes OR TOTAL(sales) GT 20000
```

Creating Boolean Expressions

A Boolean expression is a three-part clause that consists of two items to be compared, separated by a comparison operator. You can create a more complex Boolean expression by joining any of these three-part expressions with the AND and OR logical operators. Each expression that is connected by AND or OR must be a complete Boolean expression in itself, even when it means specifying the same variable several times.

For example, the following expression is not valid because the second part is incomplete.

```
sales GT 50000 AND LE 20000
```

In the next expression, both parts are complete so the expression is valid.

```
sales GT 50000 AND sales LE 20000
```

When you combine several Boolean expressions, the whole expression must be valid even if the truth value can be determined by the first part of the expression. The whole expression is compiled before it is evaluated, so when there are undefined variables in the second part of a Boolean expression, you will get an error.

Use the `NOT` operator, with parentheses around the expression, to reverse the sense of a Boolean expression.

The following two expressions are equivalent.

```
district NE 'BOSTON'
NOT(district EQ 'BOSTON')
```

Example 4–3 Using Boolean Comparisons

The following example shows a report that displays whether sales in Boston for each product were greater than a literal amount.

```
LIMIT time TO FIRST 2
LIMIT geography TO 'BOSTON'
REPORT DOWN product ACROSS time: f.sales GT 7500
```

This `REPORT` command returns the following data.

```
CHANNEL: TOTALCHANNEL
GEOGRAPHY: BOSTON
               ---F.SALES GT 7500---
               --------TIME---------
PRODUCT          JAN02      FEB02
-------------- ---------- ----------
PORTAUDIO              no         no
AUDIOCOMP             yes        yes
TV                     no         no
VCR                    no         no
CAMCORDER             yes        yes
AUDIOTAPE              no         no
VIDEOTAPE             yes        yes
```

Comparing NA Values in Boolean Expressions

When the data you are comparing in a Boolean expression involves an NA value, a YES or NO result is returned when that makes sense. For example, if you test whether an NA value is equal to a non-NA value, then the result is NO. However, if the result would be misleading, then NA is returned. For example, testing whether an NA value is less than or greater than a non–NA value gives a result of NA.

The following table shows the results of Boolean expressions involving NA values, which yield non-NA values.

Expression	Result
NA EQ NA	YES
NA NE NA	NO
NA EQ non-NA	NO
NA NE non-NA	YES
NA AND NO	NO
NA OR YES	YES

Controlling Errors When Comparing Numeric Data

If you get unexpected results when comparing numeric data, then there are several possible causes to consider:

- One of the numbers you are comparing may have a small decimal part that does not show in output because of the setting of the DECIMALS option.
- You are comparing two floating point numbers and at least one number is the result of an arithmetic operation.
- You have mixed SHORTDECIMAL and DECIMAL data types in a comparison.

Oracle Corporation recommends that you use the ABS and ROUND functions to do approximate tests for equality and avoid all three causes of unexpected comparison failure. When using ABS or ROUND, you can adjust the absolute difference or the rounding factor to values you feel are appropriate for your application. If speed of calculation is important, then you will probably want to use the ABS rather than the ROUND function.

Controlling Errors Due to Numerical Precision

Suppose `expense` is a decimal variable whose value is set by a calculation. If the result of the calculation is 100.000001 and the number of decimal places is two, then the value will appear in output as 100.00. However, the output of the following command returns NO.

```
SHOW expense EQ 100.00
```

You can use the ABS or the ROUND function to ignore these slight differences when making comparisons.

Controlling Errors When Comparing Floating Point Numbers

A standard restriction on the use of floating point numbers in a computer language is that you cannot expect exact equality in a comparison of two floating point numbers when either number is the result of an arithmetic operation. For example, on some systems, the following command returns a NO instead of the expected YES.

```
SHOW .1 + .2 EQ .3
```

When you deal with decimal data, you should not code direct comparisons such as the one above. Instead, you can use the ABS or the ROUND function to allow a tolerance for approximate equality. For example, either of the following two commands will produce the desired YES.

```
SHOW ABS((.1 + .2) - .3) LT .00001
SHOW ROUND(.1 + .2) EQ ROUND(.3, .00001)
```

Controlling Errors When Comparing Different Numeric Data Types

You cannot expect exact equality between SHORTDECIMAL and DECIMAL or NUMBER representations of a decimal number with a fractional component, because the DECIMAL and NUMBER data types have more significant digits to approximate fractional components that cannot be represented exactly.

Suppose you define a variable with a SHORTDECIMAL data type and set it to a fractional decimal number, then compare the SHORTDECIMAL number to the fractional decimal number, as shown here.

```
DEFINE sdvar SHORTDECIMAL
sdvar = 1.3
SHOW sdvar EQ 1.3
```

The comparison is likely to return NO. What happens in this situation is that the literal is automatically typed as DECIMAL and converts the SHORTDECIMAL variable

sdvar to DECIMAL, which extends the decimal places with zeros. A bit-by-bit comparison is then performed, which fails. The same comparison using a variable with a DECIMAL or a NUMBER data type is likely to return YES.

There are several ways to avoid this type of comparison failure:

- Do not mix the SHORTDECIMAL with DECIMAL or NUMBER types in comparisons. To avoid mixing these two data types, you should generally avoid defining variables with decimal components as SHORTDECIMAL.
- Use the ABS or ROUND function to allow for approximate equality. The following commands both produce YES.

```
SHOW ABS(sdvar - 1.3) LT .00001
SHOW ROUND(sdvar, .00001) EQ ROUND(.3, .00001)
```

Comparing Dimension Values

Values are not compared in the same dimension based on their textual values. Instead, Oracle OLAP compares the positions of the values in the default status of the dimension. This allows you to specify commands like the following command.

```
REPORT district LT 'SEATTLE'
```

Commands are interpreted such as these using the process below.

1. The text literal 'SEATTLE' is converted to its position in the district default status list of the dimension.
2. That position is compared to the position of all other values in the district dimension.
3. As shown by the following report, the value YES is returned for districts that are positioned before SEATTLE in the district default status list of the dimension, and NO for SEATTLE itself.

```
REPORT 22 WIDTH district LT 'SEATTLE'

DISTRICT       DISTRICT LT 'SEATTLE'
-------------- ----------------------
BOSTON                            YES
ATLANTA                           YES
CHICAGO                           YES
DALLAS                            YES
DENVER                            YES
SEATTLE                            NO
```

A more complex example assigns increasing values to the variable quota based on initial values assigned to the first six months. The comparison depends on the position of the values in the month dimension. Because it is a time dimension, the values will be in chronological order.

```
quota = IF month LE 'JUN02' THEN 100 ELSE LAG(quota, 1, month)* 1.15
```

However, if you compare values from different dimensions, such as in the expression region lt district, then the only common denominator is TEXT, and text values are compared, not dimension positions.

See Also: "Conditional Expressions" on page 4-29 for information about IF...THEN...ELSE syntax.

Comparing Dates

You can compare two dates with any of the Boolean comparison operators. For dates, “less” means before and “greater” means after. The expressions being compared can include any of the date calculations discussed in "Numeric Expressions" on page 4-15. For example, in a billing application, you can determine whether today is 60 or more days after the billing date in order to send out a more strongly worded bill.

```
bill.date + 60 LE SYSDATE
```

Dates also have a numeric value. You can use the TO_NUMBER and TO_DATE functions to change dates to integers and integers to dates for comparison.

Comparing Text Data

When you compare text data, you must specify the text exactly as it appears, with punctuation, spaces, and uppercase or lowercase letters. A text literal must be enclosed in single quotes. For example, this expression tests whether the first letter of each employee’s name is greater than the letter “M.”

```
EXTCHARS(employee.name, 1, 1) GT 'M'
```

You can compare TEXT and ID values, but they can only be equal when they are the same length. When you test whether a text value is greater or less than another, the ordering is based on the setting of the NLS_SORT option.

You can compare numbers with text by first converting the number to text. Ordering is based on the values of the characters. This can produce unexpected

results because the text is evaluated from left to right. For example, the text literal '1234' is greater than '100,999.00' because '2', the second character in the first text literal, is greater than '0', the second character in the second text literal.

Suppose name.label is an ID variable whose value is '3-Person' and name.desc is a TEXT variable whose value is '3-Person Tents'.

The result of the following SHOW command will be NO.

```
SHOW name.desc EQ name.label
```

The result of the following commands will be YES.

```
name.desc = '3-Person'
SHOW name.desc EQ name.label
```

Comparing a Text Value to a Text Pattern

The Boolean operator LIKE is designed for comparing a text value to a text pattern. A text value is *like* another text value or pattern when corresponding characters match.

Besides literal matching, LIKE lets you use wildcard characters to match more than one character in a string:

- An underscore (_) character in a pattern matches any single character.
- A percent (%) character in a pattern matches zero or more characters in the first string.

For example, a pattern of %AT_ matches any text that contains zero or more characters, followed by the characters AT, followed by any other single character. Both 'DATA' and 'ERRATA' will return YES when LIKE is used to compare them with the pattern %AT_.

The results of expressions using the LIKE operator are affected by the settings of the LIKECASE and LIKENL options. See the entries in the OLAP DML Reference for these options, both for examples of their effect on the LIKE operator and for general examples of the use of the LIKE operator.

No negation operator exists for LIKE. To accomplish negation, you must negate the entire expression. For example, the result of the following command is NO.

```
SHOW NOT ('BOSTON' LIKE 'BO%')
```

Comparing Text Literals to Relations

You can also compare a text literal to a relation. A relation contains values of the related dimension and the text literal is compared to a value of that dimension. For example, `region.district` holds values of `region`, so you can do the following comparison.

```
region.district EQ 'WEST'
```

Conditional Expressions

A conditional expression is an expression you can use to select one of two values based on a Boolean condition. A conditional expression contains the conditional operator `IF...THEN...ELSE` and has the following format.

```
IF Boolean-expression THEN expression1 ELSE expression2
```

You can use a conditional expression as part of any other expression as long as the data type is appropriate.

Note: Do not confuse a conditional expression with the `IF` command, which has similar syntax but a different purpose. The `IF` command does not have a data type and is not evaluated like an expression.

A conditional expression is processed by first evaluating the Boolean expression; then:

- If the result of the Boolean expression is `TRUE`, then ***expression1*** is evaluated and returns that value.
- If the result of the Boolean expression is `FALSE`, then ***expression2*** is evaluated and returns that value.

The *`expression1`* and *`expression2`* arguments are any valid OLAP DML expressions that evaluate to the same basic data type. However, when the data type of either value is `DATE`, it is possible for the other value to have a numeric or text data type. Because both data types are expected to be `DATE`, it will convert the numeric or text value to a `DATE`. The data type of the whole expression is the same as the two expressions.

If the result of the Boolean expression is `NA`, then `NA` is returned.

Example 4–4 Report with Conditional Expression

This example shows a sales bonus report. The bonus is 5 percent of the amount that sales exceeded budget, but if sales in the district are below budget, then the bonus is zero.

```
LIMIT month TO 'JAN02' TO 'JUN02'
LIMIT product TO 'TENTS'
REPORT DOWN district IF sales-sales.plan LT 0 THEN 0
       ELSE .05*(sales-sales.plan)

PRODUCT: TENTS
        ---IF SALES-SALES.PLAN LT 0 THEN 0 ELSE .05*(SALES-SALES.PLAN)---
          ----------------------MONTH------------------------------
DISTRICT   JAN02    FEB02    MAR02     APR02    MAY02    JUN02
--------- -------- -------- -------- ------- --------- ----------
BOSTON      229.53     0.00     0.00    0.00    584.51     749.13
ATLANTA       0.00     0.00     0.00  190.34    837.62   1,154.87
CHICAGO       0.00     0.00     0.00   84.06    504.95     786.81
.
.
.
```

Substitution Expressions

A substitution expression allows you to substitute the value of the expression for the expression itself in a command or function.

To construct a substitution expression, use an ampersand character (&) at the beginning of an expression. Using an ampersand (that is, the substitution operator) this way is also called ampersand substitution. The ampersand specifies that the expression should be evaluated with the ampersand and substitute the resulting value before it evaluates the rest of the expression.

Ampersand substitution gives you a level of indirection when you are specifying an expression. For example, when you specify an ampersand followed by a variable that holds the name of another variable, the value of the expression becomes the data in the second variable. Ampersand substitution lets you write more general programs that can operate on data that is chosen when the program is run.

You cannot use ampersand substitution in model equations.

Note: Although ampersand substitution lets you write general programs that can handle different variables and data, program lines that use ampersand substitution are executed less efficiently. Lines with ampersand substitution are not compiled; instead these lines are interpreted when the program runs. To avoid ampersand substitution, you can use the `IF` or `SWITCH` command instead.

See Also: "Controlling the Flow of Execution" on page 7-14 for information about writing conditional commands.

Example 4–5 Using Ampersand Substitution

Suppose you have a variable called `curname` that holds the name of one of the dimensions in the analytic workspace (`product`). If you execute the following command, then `REPORT` produces the single value, `product`, which is the actual value stored in the `curname` variable, as shown below.

```
report curname

CURNAME
----------
PRODUCT
```

However, if you execute the following command, then `REPORT` produces the values of the dimension `product`, as shown below.

```
report &curname

PRODUCT
--------------
TENTS
CANOES
RACQUETS
SPORTSWEAR
FOOTWEAR
```

Working with NA Values

There are cases in which you might specify an operation for which no data is available. For example, there might be no appropriate value for a given cell in a variable, for the return value of a function, or for the value of an expression that includes an arithmetic operator. In these cases, an NA (Not Available) value is automatically supplied.

NA is the value of any cell to which a specific data value has not been assigned or for which data cannot be calculated. An NA value has no specific data type.

Certain functions (for example, the aggregation functions) return an NA value when the information that is requested with the function is not available or cannot be calculated. Similarly, an expression whose value cannot be calculated has NA as its value.

To set the value of a variable or relation to NA, you can use the = command, as shown in the following example.

```
sales = NA
```

If sales is a dimensioned variable, then the = command loops through all of the values of sales, setting them to NA.

Controlling how NA values are treated

The following options and functions control how NA values are treated in expressions:

- Using the PROPERTY command, you can set the value of the NATRIGGER property on a dimensioned variable so that when a cell of the variable that contains an NA value is read, the value of the NATRIGGER expression is substituted for the NA value. You can use this substitution to increase the efficiency of some kinds of calculations and to eliminate the need for some formula objects.
- The following options control how NA values are treated in aggregation functions and in arithmetic operations with the addition (+) and subtraction (–) operators.
  - The NASKIP option controls how NA values are treated in aggregation functions.
  - The NASKIP2 option controls how NA values are treated in arithmetic operations with the addition (+) and subtraction (–) operators.

- The NAFILL function returns the values of the source expression with any NA values appearing as the specified fill expression. You can include this function in an expression to control the format of its value.

Working with the NATRIGGER Property

An NATRIGGER property expression is evaluated before applying the NAFILL function or the NASKIP, NASKIP2, or NASPELL options. If the NATRIGGER expression is NA, then the NAFILL function and the NA options have an effect. Additionally, the NATRIGGER property allows you a good deal of flexibility about handling NA values:

- You can make NA triggers recursive or mutually recursive by including triggered objects within the value expression. You must set the RECURSIVE option to yes before a formula, program, or other NATRIGGER expression can invoke a trigger expression again while it is executing. For limiting the number of triggers that can execute simultaneously, see the TRIGGERMAXDEPTH option.
- You can replace the NA value in the cells of the variable with the NATRIGGER expression value by setting the TRIGGERSTOREOK option to yes and setting the STORETRIGGERVAL property on the variable to yes.

The ROLLUP and AGGREGATE commands and the AGGREGATE function ignore the NATRIGGER property setting for a variable during a rollup or aggregation operation. Additionally, the NATRIGGER property expression on a variable is not evaluated when the variable is simply exported with an EXPORT TO EIF file command. The NATRIGGER property expression is only evaluated if the variable is part of an expression that is calculated during the export operation.

Using NASKIP

The NASKIP option controls how NA values are treated in aggregation functions.

- By default, the NASKIP option is set to YES, and NA values are ignored by aggregation functions. Only expressions with actual values are used in calculations.
- If you set the NASKIP option to no, then NA values are considered as input to aggregation functions. If any of the values being considered are NA, then the function returns NA for that value.

Setting NASKIP to no is useful for cases in which having NA values in the data makes the calculation itself invalid. For example, when you use the MOVINGMAX function, you specify a range from which to select the maximum value.

- If NASKIP is YES (the default), then MOVINGMAX returns NA only when all the values in the range are NA.
- If NASKIP is NO and any value in the range is NA, then MOVINGMAX returns NA.

Using NASKIP2

The NASKIP2 option controls how NA values are treated in arithmetic operations with the addition (+) and subtraction (-) operators.

- By default, the value of the NASKIP2 option is NO. NA values are treated as NAs in arithmetic operations using the addition (+) and subtraction (-) operators. If any of the operands being considered is NA, then the arithmetic operation evaluates to NA. For example, by default, 2+NA results in NA.
- If you set the value of the NASKIP2 option to yes, then zeroes are substituted for NA values in arithmetic operations using the addition (+) and subtraction (-) operators. The two special cases of NA+ NA and NA-NA both result in NA.

Using NAFILL

NASKIP and NASKIP2 do not change your data. They only affect the results of calculations on your data. If you would prefer a more targeted influence on any kind of expressions, and want the option of making an actual change in your data, then you can use the NAFILL function.

The effect of the NAFILL function is limited to the single expression you specify. It can be any kind of expression, not just a function or an addition (+) or subtraction (-) operation. In addition, you can use NAFILL to substitute anything for the NAs in the expression, not just zeroes. Moreover, using assignment statements, you can use NAFILL to make a permanent substitution for NAs in your data.

NAFILL returns the value of a specified expression unless its value is NA, in which case NAFILL returns the substitute value you specify.

The following command uses NAFILL to replace the NA values in the sales variable with the number 1 and then assign those values to the variable. This makes the substitution permanent in your data.

```
sales = NAFILL(sales, 1)
```

The following command illustrates the use of NAFILL for more specialized purposes. By substituting zeros for NA values, NAFILL in this example forces the AVERAGE function to include NA values when it counts the number of values it is

averaging. The substitution is temporary, lasting only for the duration of this command.

```
SHOW AVERAGE(NAFILL(sales 0.0) district)
```

5

Populating Workspace Data Objects

This chapter provides an overview of how to populate workspace data objects that hold source data and how to populate variables with calculated values. It includes the following topics:

- Overview: Populating an Analytic Workspace
- Maintaining Dimensions and Composites
- Assigning Values to Data Objects
- Calculating and Analyzing Data

Overview: Populating an Analytic Workspace

To use an analytic workspace, there must be data in it. There are two basic types of data: fact data and dimensions. Fact data is stored in **variable** workspace objects; dimensions, containing dimension values, are stored in **dimension** workspace objects.

Variables and dimensions can be populated:

- By loading data from relational tables. For example, you might load sales fact data into a variable from a sales fact table, load time dimension values from a time dimension table, customer dimension values from a customer dimension table, and product dimension values from a product dimension table.
- As the result of a calculation. For example, a sales forecast variable might be populated using the results of a forecasting function.
- By loading data from flat files using data loaders controlled through the OLAP DML.
- Manually, although this method is typically used only to enter a small number of values.

To explicitly populate data objects in an analytic workspace, take the following steps:

1. Specify the values for each dimension. These values provide indexes to the actual data, which is stored in analytic workspace variables.
2. Specify the values for each relation. These values indicate the relationships between dimensions.
3. For variables that provide the source data for your application, specify the actual data values.

You can populate an analytic workspace using programs written using the SQL command and data loading commands. The OLAP DML commands that you typically use to populate data objects are listed in the following table.

Command	Description
= ***or*** SET	Assigns the results of an expression to a variable, option, or relation. For more information, see "Assigning Values to Data Objects" on page 5-10 and "Using Models to Calculate Data" on page 8-2.
MAINTAIN	Adds, deletes, renames, moves, or merges values in a dimension; and adds, deletes, and merges values in a composite. For more information, see "Maintaining Dimensions and Composites" on page 5-3.
FILEREAD	Stores the data that is read from an input file into a dimension, composite, relation, or variable. For more information, see Chapter 11, "Reading Data from Files".
SQL	Retrieves data from relational tables into a dimension or variable. For more information, see Chapter 10, "Working with Relational Tables".
IMPORT	Copies workspace data and definitions from an EIF file.

Maintaining Dimensions and Composites

The first step in populating an analytic workspace is to store values in analytic workspace dimensions. The list of stored dimension values is called the default status list of the dimension. When you first attach an analytic workspace, the default status list is the current status list of each dimension.

Using the MAINTAIN command, you can add, delete, merge, reposition, or change simple, composite, or conjoint dimension values, and you can reposition concat dimension values. Storing and manipulating the values of a dimension is called maintaining the dimension.

How Maintaining a Dimension Affects Dimension Status

As outlined in the following table, using the MAINTAIN command sometimes affects dimension status.

IF you use the MAINTAIN command with . . .	THEN . . .
the ADD, DELETE, MERGE, or MOVE keyword and the current status of a dimension is not ALL,	the dimension status is reset to ALL before it performs the requested maintenance.
a dimension that has a pushed status list (that is, a status list that was created using the PUSH commands),	the pushed status list of the dimension is cleared, and popping that dimension has no effect.

For more information on popping and pushing dimension status, see "Introducing Dimension Status" on page 6-2.

Avoiding Deferred Maintenance

When you maintain a dimension, the objects that are dimensioned by it must be modified. If these objects are in memory, then they are modified immediately; if these objects are *not* in memory, then maintenance is deferred until they are loaded into memory.

In situations that involve a lot of dimension maintenance and a large update at the end, deferred maintenance can trigger errors. Examples are issuing a MAINTAIN DELETE ALL command, or performing a data load in which a large number of values is added to a dimension. Before starting such projects, load into memory the objects that use that dimension so that deferred maintenance is unnecessary. You can do this by using commands similar to the following, where the sample dimension is product.

```
LIMIT NAME TO OBJ(ISBY product)
LOAD &values(NAME)
MAINTAIN product ADD ...
```

Adding Values to Dimensions

To add new values to the end of a dimension or composite, use the MAINTAIN command with the ADD keyword. The actual way that the values are added, and the arguments that you use vary depending on whether you are adding values to a dimension or a composite.

You do not add values directly to a concat dimension. Instead, if you add a value to a base dimension of the concat dimension, then Oracle OLAP automatically adds the value to the concat dimension. Similarly, you do not add values to a dimension surrogate, but if you add a value to the dimension of the dimension surrogate, then you can add a surrogate for the new value to the dimension surrogate.

You can use the `MAINTAIN` command with the `MERGE` keyword as a quick way to make sure all dimension values on a separate list are included in a dimension. When you use this syntax, the new values from the list are automatically added and the duplicates are ignored. This method of entering dimension values can save a significant amount of time when you have a large number of values to enter.

You can use the `MAINTAIN` command with the `ADD` keywords to add values to a dimension in the following ways:

- You can merely specify the values that you want to add. In this case, the values are added to the end of the list of dimension values.
- You can specify both the values that you want to add and where you want the values to be placed.

Example 5–1 Adding Values to Dimensions

This command adds `ATLANTA` at the beginning of the list of cities and inserts `PEORIA` after `OMAHA`.

```
MAINTAIN city ADD 'ATLANTA' FIRST, 'PEORIA' AFTER 'OMAHA'
```

Displaying the default status list for the `city` dimension shows that the new values have been added in the appropriate places in the list.

```
SHOW VALUES(city NOSTATUS)
ATLANTA
CONCORD
LINCOLN
NEW YORK
OMAHA
PEORIA
SEATTLE
```

Updating Relations When Merging New Values

When you are merging values into a dimension it is a good practice to update any relations that involve that dimension:

- In some cases, using the simplified syntax of the `MAINTAIN` command shown below, you can update a relation at the same time you merge values into a dimension.

  ```
  MAINTAIN dimension MERGE [exp [RELATE relation] ]
  ```

  The *exp* argument specifies a dimensioned expression whose values you want to merge into the dimension; for example, the name of a dimensioned text variable that contains dimension values.

  The `RELATE` *relation* phrase specifies the name of the relation that you want to update.

  Note: The *exp* argument must be dimensioned and at least one of these dimensions must also be in the definition of the relation that is specified in the `RELATE` *relation* phrase.

- In other cases, you need to explicitly update any relations that involve that dimension.

For information about explicitly updating relations, see "Assigning Values to Data Objects" on page 5-10.

Suppose you want to define a composite, named `comp_proddist`, that is made up of all combinations of the first three values of the `product` dimension and the first five values of the `district` dimension. You can efficiently include all 15 values with the following commands.

```
DEFINE comp_proddist COMPOSITE <product district>
LIMIT product TO FIRST 3
LIMIT district TO FIRST 5
MAINTAIN comp_proddist MERGE <product district>
```

This method works with conjoint dimensions as well.

Deleting Values from Dimensions

You can use the `MAINTAIN` command with the DELETE keyword to remove values from a dimension. You select the values that you want to delete in much the same way that you select values using the LIMIT command. You can select for deletion:

- One value, a list of values, a range of values, or all values
- The values that match a list of values of a named related dimension
- The values that are first, last, or in a specified position in the dimension
- The values that meet a Boolean criterion
- After the dimension values are sorted according to a specified criterion, the top or bottom *n* values of the dimension, or the top or bottom *n* performers, by percentage
- For a hierarchical dimension, the values that have a certain relationship within the hierarchy
- The values in the dimension that match the values in a valueset

Example 5–2 Deleting Values from a Dimension

Suppose that you want remove from `city` all those cities with a population of less than 75,000 people. Before you issue the command, the default status list for the `city` dimension contains the six values shown below.

```
SHOW VALUES (city NOSTATUS)
ATLANTA
CONCORD
LINCOLN
COLUMBUS
PEORIA
SEATTLE
```

You use the variable `population.c`, which contains the population for each city.

```
MAINTAIN city DELETE population.c LT 75000
```

Assuming that only Lincoln and Peoria have populations of fewer than 75,000, the default status list of the `city` dimension now contains the following values.

```
SHOW VALUES (city NOSTATUS)
ATLANTA
CONCORD
```

```
COLUMBUS
SEATTLE
```

Deleting Values from Conjoint Dimensions

You can use the MAINTAIN command with the DELETE keyword to delete values from a conjoint dimension.

You can also delete values from a conjoint dimension by using the MAINTAIN command directly on the base dimension of the conjoint dimension. When you delete a value from the base dimension, any values associated with that base dimension value are deleted from the conjoint dimension.

Suppose you have a conjoint dimension named prod_dist with the base dimensions of product and district. To delete the value <'SNOWSHOES' 'ATLANTA'> from that conjoint dimension, you would use the following command.

```
MAINTAIN prod_dist DELETE <'SNOWSHOES' 'ATLANTA'>
```

Changing the Position of Dimension Values

You can use the MAINTAIN command with the MOVE keyword to change the position of one or more values in a dimension list. You cannot change the position of a value in a time dimension or in a composite.

When you want to store the dimension values in alphabetical order, you can first use the SORT command to temporarily sort the values, and then use the MAINTAIN command to store the values in the sorted order.

Use the TEXT variable textvar to move SEATTLE to the end of the list of cities.

```
textvar = 'SEATTLE'
MAINTAIN city MOVE textvar LAST
```

Storing Dimension Values in Sorted Order

You can store the values of a dimension in sorted order by taking the following actions:

1. Limit the dimension to all of its values.

   ```
   LIMIT dimension TO ALL
   ```

2. Sort the dimension values based on your desired sorting criterion.

   ```
   SORT dimension A sort-criterion
   ```

   To sort the values alphabetically, sort by the dimension itself.

3. Store the dimension values in their sorted order.

   ```
   MAINTAIN dimension MOVE VALUES(dimension) FIRST
   ```

Suppose that the default status list for the `city` dimension contains the following values.

```
SHOW VALUES (city NOSTATUS)
ATLANTA
CONCORD
LINCOLN
COLUMBUS
PEORIA
SEATTLE
```

The following commands sort the values of `city` in alphabetical order and then store the values in that order.

```
SORT city A city
MAINTAIN city MOVE VALUES(city) FIRST
```

The default status list of `city` reflects the new sorted order.

```
SHOW VALUES (city NOSTATUS)
ATLANTA
COLUMBUS
CONCORD
LINCOLN
PEORIA
SEATTLE
```

Maintaining Composites and Conjoint Dimensions

Both composites and conjoint dimensions are lists of dimension-value combinations in which one value is taken from each of the dimensions on which the composite or conjoint dimension is based. Composites and conjoint dimensions differ in the way that they are maintained.

Maintaining Composites

Composites are internal structures that are automatically maintained. Consequently, the simplest way to maintain a composite is to merely maintain its base dimensions and let the values in the composite be maintained automatically.

In most cases, it is not necessary to do anything to maintain composites. However, if you want to have a very fine degree of control, you may have to explicitly maintain the composite. In this case, you can use the MAINTAIN command to add, delete, and merge values.

Maintaining Conjoint Dimensions

Conjoint dimensions, unlike composites, are actual dimensions that you must explicitly maintain using the MAINTAIN command.

Maintaining Concat Dimensions

You can use the MAINTAIN command to change the order of the values in a concat dimension. If you use the MAINTAIN MOVE command on a simple dimension that is a component of a concat dimension, then the positions of the values of the concat dimension are not affected.

You cannot use the MAINTAIN command to add, delete, or rename concat dimension values or merge values from another dimension to those of the concat dimension.

If you use the MAINTAIN command to add a value to a simple dimension that is a component of a concat dimension, then Oracle OLAP adds that value to the concat dimension as a value of the component dimension. If you merge values from a simple dimension with a component simple dimension, then Oracle OLAP adds those values to the concat dimension as values of the component dimension.

If you delete or rename a value of a simple dimension that is a component of a concat dimension, then Oracle OLAP deletes or renames the value in the concat dimension. If you use the MAINTAIN command to add, merge, or delete the values of a simple dimension component of a concat dimension, the status of the concat dimension is automatically set to ALL.

Assigning Values to Data Objects

An expression creates temporary data; you can display the resulting values, but these values are not automatically saved in your analytic workspace. If you want to save the result of an expression, then you store it in an object that has the same data

type and dimensions as the expression. You use an assignment statement to store the value that is the result of the expression in the object.

An assignment statement is composed of the OLAP DML = operator that is preceded by an expression (on the left) and followed by an expression (on the right).

```
target-expression = source-expression
```

The assignment statement sets the value of the target expression equal to the results of the source expression.

> **See Also:** Chapter 3, "Defining Data Objects" for information about how data is stored in data objects.

Using Objects in Assignment Statements

The following table outlines the objects that you can use in assignment statements and indicates whether you can use them as a target or source expression.

Object	Target Expression	Source Expression
Variable	Yes	Yes
Relation	Yes	Yes
Dimension	*Only* in models	Yes
Surrogate	No	Yes
Composite	No	Yes
Worksheet	Yes	Yes
Function	No	Yes
Formula	No	Yes
Valueset	No	Yes

When you use the = operator to assign the value of a single-cell expression to a single cell, a single value is stored. However, when you use the = operator to assign the value of a single-cell expression to a target variable that has one or more dimensions, then the assignment loops over the values in status for each dimension of the target variable and assigns a data value to the corresponding cells of the variable.

Example 5–3 Assigning Values to Variables

The `choicedesc` variable is dimensioned by `choice`. Before you enter data for the variable, the cells of the variable contain only NA values.

```
CHOICE           CHOICEDESC
-------------- --------------------
REPORT         NA
GRAPH          NA
ANALYZE        NA
DATA           NA
QUIT           NA
```

Suppose you initialize the `choicedesc` variable using the following command.

```
choicedesc =  JOINCHARS ('Description for ' choice)
```

Now all of the `choicedesc` cells of the variable contain the appropriate values.

```
CHOICE             CHOICEDESC
-------------- -------------------------
REPORT         Description for REPORT
GRAPH          Description for GRAPH
ANALYZE        Description for ANALYZE
DATA           Description for DATA
QUIT           Description for QUIT
```

The next example shows an expression that is dimensioned by `time`, `product`, and `district` and is assigned to a new variable. The expression calculates a 2002 sales plan based on unit sales in 2001.

```
DEFINE units.plan INTEGER <month product district>
LIMIT month TO 'DEC02'
units.plan = LAG(units 12 month) * 1.15
```

How Values Are Assigned to Variables with Composites

When assigning data to variables with composites, the source expression is evaluated for every combination of the dimension values in status for the target variable, including combinations of the sparse dimensions for which the target variable currently has no cells. If the source expression is not NA for those combinations where the target currently has no cells, then new cells are created and the data is assigned to them.

When you use the = command to assign values to a target variable that has a composite, the command does the following automatically:

- Creates any missing target variable cells that are being assigned non-NA values.
- Adds to the composite all the dimension-value combinations that correspond to those new cells.

Thus, both the target variable and the composite might be larger after an assignment. If you want to assign values only to cells that already exist in the target variable, then use the `ACROSS` keyword in the = command.

The OLAP DML gives you the ability to specify a different evaluation behavior when it assigns data to variables with composites. You can alter the default evaluation behavior of the assignment statement so that the source expression is evaluated only for those combinations of the dimension values in status for which the target variable currently has cells.

Because the composite of the sparse dimension is what keeps track of which combinations of the sparse dimensions have data cells, you use the following syntax to specify this different evaluation behavior.

```
varname = expression ACROSS composite
```

The *`varname`* argument is the name of the variable. It is the target to which the data is assigned.

The *`expression`* argument is the source expression that holds the data that will be assigned to the target variable.

The `ACROSS` keyword indicates that you want to alter the default evaluation behavior and cause the evaluation of the composite of the target variable.

The *`composite`* argument is the composite for the sparse dimensions on the target variable. If the variable was defined with a named composite, then specify the name of the composite. If the variable was defined with an unnamed composite, then use the `SPARSE` keyword to refer to the unnamed composite (for example, `SPARSE <MARKET PRODUCT>`).

Example 5–4 Assigning Values to Variables with Composites

To have data assigned from `sales` only into existing data cells of `sparse_sales`, whose associated dimension values are in status, use the following command.

```
sparse_sales = sales ACROSS SPARSE<product market>
```

The ACROSS keyword is particularly helpful when the source expression is a single value. If there are no limits on the dimensions of sparse_sales, then an assignment command like the following will create cells for every combination of dimension values because there are no cases where the source expression is NA.

```
sparse_sales = 0
```

This defeats the purpose of a sparse variable.

In contrast, the following command will set only existing cells of sparse_sales to 0.

```
sparse_sales = 0 ACROSS SPARSE<product market>
```

Assigning Values to Relations

You can assign values to a relation using an assignment statement. When executing the assignment statement, a loop is performed over the values in status for each dimension of the target relation and assigns a data value to the corresponding cell of the target relation.

You can assign values to a relation with a text dimension by assigning one of the following:

- A text value of the dimension.
- An integer that represents the position of the dimension value in the default status list of the dimension.

Assigning Values to Dimensions

In most cases, you cannot use an assignment statement to assign values to dimensions. However, in model equations, if the result of a calculation is numeric, then you can use the = operator to assign the results to a dimension value. However, equations (that is, expressions) in models differ in several ways from expressions used in other contexts.

See Also: Chapter 8, "Working with Models" for more information on working with models.

Assigning Values to Specific Cells of a Data Object

You can use a QDR with the target of an assignment statement. This lets you assign a value to specific cells in a variable or relation.

The following example assigns the value 10200 to the data cell of the sales variable that is specified in the qualified data reference. If the variable named sales does not already have a value in the cell associated with BOSTON, TENTS, and JAN99, then the value is assigned to the cell and thus it is added to the variable. If a value already exists in the cell, the value 10200 overwrites the previous value.

```
sales(market 'BOSTON' product 'TENTS' month 'JAN99')= 10200
```

See Also: "Specifying a Single Value for the Dimension of an Expression" on page 4-6 for information about QDRs.

Calculating and Analyzing Data

Typically, using the OLAP DML, you calculate and analyze data in the following ways:

- Perform common calculations using built-in functions that are described in detail in the Oracle9i OLAP DML Reference help.
- Aggregate (or roll up) data in variables that are dimensioned by one or more hierarchical dimensions as outlined in Chapter 12, "Aggregating Data".
- Allocate data to a variable from a source object based on the data of a base object as described in Chapter 9, "Allocating Data".
- Create populated solution variables using the MODEL object as described in Chapter 8, "Working with Models".
- Forecast data based on analysis of trends as described in the entries for the FCSET, FCOPEN, FCEXEC, FCCLOSE, and FCQUERY commands in the Oracle9i OLAP DML Reference help.

The OLAP DML provides built-in functions for numeric analysis. The categories of these functions are described below.

Category	Description
Numeric cell-by-cell	Operate on each cell of an expression or variable.
Time series	Retrieve values from a previous or future time period and perform calculations on those values.
Statistical	Perform calculations for statistical analysis.
Financial	Perform calculations for financial analysis.
Aggregation	Return an aggregate value, generally consisting of a single value for many values of the input expression.

See Also:

- Oracle9i OLAP DML Reference help for a categorized list of functions.
- "Numeric Expressions" on page 4-15 for information on working with numeric expressions.

6 Selecting Data

This chapter introduces dimension status and the use of the LIMIT command to temporarily change your view of the data in an analytic workspace. The LIMIT command is equivalent to the WHERE clause of a SQL SELECT statement.

This chapter includes the following topics:

- Introducing Dimension Status
- Limiting to a Simple List of Values
- Limiting Using a Boolean Expression
- Limiting to the Top or Bottom Values
- Limiting to the Values of a Related Dimension
- Limiting Based on the Position of a Value in a Dimension
- Limiting Based on a Relationship Within a Hierarchy
- Limiting Composites and Conjoint Dimensions
- Ways of Limiting Conjoint Dimensions
- Limiting Concat Dimensions
- Working with Null Status
- Working with Valuesets

Introducing Dimension Status

The **current status list** of a dimension is an ordered list of currently accessible values for the dimension. Values that are in the current status list of a dimension are said to be "in status." The current status list of a dimension determines the selection of the data from all of the objects that are dimensioned by it.

For dimensions, only those dimension values that are in the current status list are accessed. For dimensioned objects, only those data values that are indexed by dimension values in the current status list are accessed.

As a loop is performed through a dimensioned object, the order of the dimension values in the current status list is used to determine the order in which the values of the object are accessed.

Whether or not a dimension value is in status merely restricts your view of the value during a given session; it does not permanently affect the values that are stored in the analytic workspace.

When you first attach an analytic workspace, the current status list of each dimension consists of all of the values of the dimension that have read permission, in the order in which the values are stored. This list of values is called the **default status list** for the dimension.

A status list of a dimension surrogate is the same as the status list of its dimension. A surrogate does not have a current or default status list separate from its dimension.

Changing the Current Status List

You can change the current status list for a dimension by using:

- The `LIMIT` command to change the values and the order of the values in the current status list of a dimension.
- The `SORT` command to arrange the order of values in the current status list of a dimension.

Changing the Default Status List

You can change the default status list of a dimension in the following ways:

- You can add, delete, move, merge, and rename values in a dimension by using the `MAINTAIN` command. However, with a concat dimension you use the `MAINTAIN` command only to move its values to a different order in the dimension.

- You can change the read permission of values that are associated with a dimension by using the PERMIT command or the PERMITRESET command.

See Also:

- "Maintaining Dimensions and Composites" on page 5-3 for information on storing and maintaining dimension values.
- "Adding Security to an Analytic Workspace" on page 2-12 for information on setting permissions on workspace objects.

Identifying and Retrieving Status Lists

You can use the following commands and functions to identify and retrieve the status of dimension values.

Command or function	Description
INSTAT function	Checks whether a dimension value is in the current status list of a dimension.
STATFIRST function	Retrieves the first value in the current status list of a dimension.
STATLAST function	Retrieves the last value in the current status list of a dimension.
STATUS command	Sends to the current outfile the status of one or more values in a dimension, or the status of all dimensions in an analytic workspace.
VALUES function	Retrieves different values depending on the keyword that you specify: ■ If you specify the NOSTATUS keyword, then the function retrieves the default status list of a dimension list. ■ If you specify the STATUS keyword, then the function retrieves the current status list of a dimension. ■ Depending on whether you specify the INTEGER keyword, the function either returns a multiline text value that contains one dimension value per line or returns, as integers, the position numbers of the dimension values.

Saving and Restoring Dimension Status

You can save the current status of a dimension in the following ways.

- If you want to save the current status (or the value of a dimension) for use in any session, then use a named valueset. Use the `DEFINE VALUESET` command to define the valueset.
- If you want to save the current status (or the value of a dimension, a valueset, an option, or a single-cell variable) for use in the current program, then use the `PUSHLEVEL` and `PUSH` commands. You can restore the current status values using the `POPLEVEL` and `POP` commands.
- If you want to save, access, or update the current status (or the value of a dimension, a valueset, an option, a single-cell variable, or a single-cell relation) for use in the current session, then use a named context. Use the `CONTEXT` command to define the context.

Contexts are the most sophisticated way to save object values for use in an analytic workspace. With contexts, you can access, update, and commit the saved object values. In contrast, `PUSH` and `POP` simply allow you to save and restore values. Typically, you only used the `PUSH` and `POP` commands within a program to make changes that apply only during the program execution.

See Also: "Preserving the Session Environment" on page 7-19 for more information about saving environment settings.

Limiting to a Simple List of Values

A common way of selecting data is to limit a dimension to a value or list of values. When limiting dimension values, you can substitute a dimension surrogate for its dimension. The simplified syntax for using the `LIMIT` command in this way is shown below.

```
LIMIT dimension TO values
```

The *`values`* argument can consist of any combination of:

- Dimension values, expressed as literal values separated by commas, or as a multiline text expression, each line of which is a value of the dimension.
- Ranges of dimension values, expressed as *`value1`* `TO` *`value2`*.

- Integer values that represent the logical positions of dimension values, expressed as comma-separated integers.
- Ranges of integer values that represent the logical positions of dimension values, expressed as `value1 TO value2`.
- Valuesets.

Suppose that you want a report of footwear sales in Boston for January through March 1995. The following commands limit the appropriate dimensions and request the report.

```
LIMIT month TO 'JAN95' 'FEB95' 'MAR95'
LIMIT product TO 'FOOTWEAR'
LIMIT district TO 'BOSTON'
REPORT sales
```

The report output looks like this.

```
DISTRICT: BOSTON
               -------------SALES--------------
               -------------MONTH--------------
PRODUCT          JAN95      FEB95      MAR95
-------------- ---------- ---------- ----------
FOOTWEAR        91,406.82  86,827.32 100,199.46
```

As an example of limiting dimension values using a dimension substitute, suppose you have a `NUMBER` dimension named `storeid` that has store identification numbers as values. The values of `storeid` are 10, 20, 30, 100, 110, 120, and 200. You have an `INTEGER` dimension surrogate for `storeid`, named `storenum`, that has an integer value for each position of the values of `storeid`. The values of `storenum` are the integers 1 through 7. You can limit the current status list of both `storeid` and `storenum` to the same set of values with any of the following commands.

```
LIMIT storeid TO 10, 100
LIMIT storenum TO 1, 4
LIMIT storenum TO storeid 10, 100
LIMIT storenum TO storenum 1, 4
```

Limiting Using a Boolean Expression

You can use the `LIMIT` command to limit a dimension according to the result of a Boolean expression. The simplified syntax for using the `LIMIT` command in this way is shown below:

```
LIMIT dimension TO Boolean-expression
```

When you use this form of the LIMIT command, the values that are currently in status are replaced with those dimension values for which the Boolean expression is true.

When you are constructing a Boolean expression, keep the following points in mind:

- The Boolean expression must be dimensioned by the dimension whose status is being set.
- The data types of the expressions you are comparing in the Boolean expression must be similar.

  For example, the following Boolean expression has similar data types on both sides of the Boolean operator GT.

  ```
  LIMIT market TO units.m GT 50000
  ```

In the following example, the values of the TOTAL function are broken out by product and compared to a literal (that is, the number 12000000). The LIMIT command replaces the values that are currently in status for the product dimension with the values of the product dimension whose sales, totaled for all months and districts, are greater than 12 million.

```
LIMIT product TO TOTAL(sales product) GT 12000000
```

How LIMIT Handles Boolean Multidimensional Expressions

An understanding of how the LIMIT command handles Boolean expressions with more than one dimension is important to the successful use of the command.

The result of a simple Boolean expression is a single value. When you use the LIMIT command with a Boolean expression, no looping is performed through the dimensions to create and return an array of values for the expression. Instead, the first value in the dimension status list is identified for each dimension in the expression, the expression using those values is evaluated, and a single value is returned.

If you want the result of the Boolean expression to have dimensionality, then use the EVERY, ANY, or NONE functions, which let you specify the dimensions of the result of the Boolean expression.

Suppose that `month`, `district`, and `product` have the dimension status shown below.

```
The current status of MONTH is:
JAN95 TO MAR95
The current status of DISTRICT is:
BOSTON
The current status of PRODUCT is:
ALL
```

Now you want products that have more than $90,000 worth of sales in at least one of the months to be in status for the `product` dimension. By issuing the following command, you can see which values in the current dimension status meet this condition.

```
REPORT sales GT 90000
```

As shown below, the report displays `YES` in both the `FOOTWEAR` and `CANOES` rows. Both of these products have sold more than $90,000 on at least one occasion during January through March 1995.

```
DISTRICT: BOSTON
               ---------SALES GT 90000---------
               -------------MONTH--------------
PRODUCT          JAN95      FEB95      MAR95
-------------- ---------- ---------- ----------
TENTS                  NO         NO         NO
CANOES                 NO         NO        YES
RACQUETS               NO         NO         NO
SPORTSWEAR             NO         NO         NO
FOOTWEAR              YES         NO        YES
```

You might think that limiting the `product` dimension using only the simple Boolean expression shown below would give you your desired result.

```
LIMIT product TO sales GT 90000
```

However, when the Boolean expression is evaluated, no looping is performed through the `sales` variable to create and return an array of values for the `product` dimension. Instead, only the first value in the dimension status list is used for each dimension in `sales` ***other*** than the `product` dimension. In this case, `JAN95` is used for the value of the `month` dimension of the `sales` variable and `BOSTON` is used for the value of the `DISTRICT` dimension.

For JAN95 and BOSTON, the Boolean expression evaluates to TRUE only for the FOOTWEAR product. Consequently, only FOOTWEAR is in status for the product dimension.

As shown below, a report of sales in Boston only displays values for the FOOTWEAR product that have sold more than $90,000 on at least one occasion during January through March 1995.

```
REPORT sales

The current status of PRODUCT is:
FOOTWEAR
DISTRICT: BOSTON
               -------------SALES--------------
               -------------MONTH--------------
PRODUCT          JAN95      FEB95      MAR95
-------------- ---------- ---------- ----------
FOOTWEAR        91,406.82  86,827.32 100,199.46
```

Limiting to Values That Match an Expression

The way to limit a dimension to *all* dimension values that match a Boolean expression is to use the ANY function with the Boolean expression.

Example 6–1 Limiting Using the ANY function

The LIMIT command (shown below) illustrates how to use the ANY function to limit the product dimension to all dimension values that have a value of more than $90,000 in the sales variable (that is, CANOES and FOOTWEAR):

- The first argument for the ANY function (that is, sales GT 90000) is the Boolean expression you want to evaluate.
- The second argument for the ANY function (that is, product) indicates the dimensionality of the result of the Boolean expression.

In this example, when the Boolean function is evaluated, a test is performed for TRUE values along the product dimension, and returns an array of values.

```
LIMIT product TO ANY(sales GT 90000, product)
```

The product dimension has both CANOES and FOOTWEAR in status. Both of these products sold more than $90,000 on at least one occasion during January through March 1995.

As shown below, a report for sales in Boston displays ***both*** the `CANOES` and `FOOTWEAR` products.

```
REPORT sales

The current status of PRODUCT is:
CANOES, FOOTWEAR
DISTRICT: BOSTON
               -------------SALES--------------
               -------------MONTH--------------
PRODUCT          JAN95      FEB95      MAR95
-------------- ---------- ---------- ----------
CANOES          66,013.92  76,083.84  91,748.16
FOOTWEAR        91,406.82  86,827.32 100,199.46
```

Limiting to the Top or Bottom Values

You can set the dimension values that are currently in status to the top or bottom performers based on a criterion represented as an expression. The simplified syntax for using the `LIMIT` command in this way is shown below:

```
LIMIT dimension TO [BOTTOM|TOP] n BASEDON expression
```

You can also set the dimension values that are currently in status to the top or bottom performers, by percentage, based on a criterion represented as an expression. The simplified syntax for using the `LIMIT` command in this way is shown below.

```
LIMIT dimension TO [BOTTOM|TOP] percent PERCENTOF expression
```

This construction sorts values based on their contribution, by percentage, to an expression and then places the identified values in status.

It can happen that the last item in status, based on a `PERCENTOF` criterion, is one of a number of dimension values having the same associated criterion value. In this case, `LIMIT` includes all dimension values with that criterion value in the resulting status, even when that causes the total of the criterion value to far exceed the specified percentage.

Note: Do not use a criterion expression that changes its own value.

Example 6–2 Limiting to the Top or Bottom Values Based on Criterion

Suppose the status list is sorted in descending order according to the values of `sales`, and only the top two performers are kept in status. Here the `TOP` and `BASEDON` keywords are used to limit the status of a dimension, using the values of a variable as a criterion.

```
LIMIT product TO 'SPORTSWEAR'
LIMIT month TO 'JUL96'
LIMIT district TO TOP 2 BASEDON sales
```

Suppose that you issue the following `REPORT` command.

```
REPORT DOWN district sales
```

The following report is produced, which shows the results of the `LIMIT` commands.

```
PRODUCT: SPORTSWEAR
               --SALES---
               --MONTH---
DISTRICT         JUL96
-------------- ----------
DALLAS         220,416.81
ATLANTA        211,666.14
```

Example 6–3 Limiting to the Top or Bottom Values Based on Percentage

Suppose you want to sort products in descending order by the contribution of each product to `TOTAL(sales)` and then add values to the status list, starting from the top, until the cumulative total of `sales` by `product` reaches or exceeds 30 percent of all sales. To limit the dimension in this way, you can use the following command.

```
LIMIT product TO TOP 30 PERCENTOF TOTAL(sales, product)
```

The following commands produce a report for January through March 2002 of products in the Boston district that reached or exceeded 30 percent of all sales.

```
LIMIT month TO 'JAN02' 'FEB02' 'MAR02'
LIMIT district TO 'BOSTON'
LIMIT product TO TOP 30 PERCENTOF TOTAL(sales, product)
REPORT sales
```

This output of the report is shown below.

```
DISTRICT: BOSTON
               -------------SALES--------------
               -------------MONTH--------------
PRODUCT          JAN02      FEB02      MAR02
-------------- ---------- ---------- ----------
FOOTWEAR        91,406.82  86,827.32 100,199.46
CANOES          66,013.92  76,083.84  91,748.16
```

Limiting to the Values of a Related Dimension

You can use the LIMIT command to limit a dimension to the values of one or more related dimensions. The simplified syntax for using the LIMIT command in this way is shown below:

```
LIMIT dimension TO reldim [reldim-val]
```

The *reldim* argument is the name of a relation or a dimension that is related to the dimension being limited. Using a relation name allows you to choose which relation is used when there is more than one.

The *reldim-val* argument is a list of values of the related dimension, and not the dimension being limited. If this argument is present in a LIMIT command, then status is obtained by selecting the values of the dimension being limited, which are related to related values. If *reldim-val* is omitted, then the current status of *reldim* is used.

Example 6–4 Limiting Using a Related Dimension

The following command limits district to BOSTON and ATLANTA, which are in the EAST region.

```
LIMIT district TO region 'EAST'
```

This command limits product to SPORTSWEAR and FOOTWEAR, which are in the division that appears last in the list of DIVISION values.

```
LIMIT product TO division LAST 1
```

How Limiting to a Related Dimension Determines Status

When you limit a dimension to a related dimension, the current status list is created in a two-step process:

1. The values in the dimension current status list are arranged in the order of the values of the related dimension.
2. If there is more than one value of the dimension for any value of the related dimension, then the values in the dimension current status list are arranged in the order of their default status list.

Suppressing the Sort When Limiting to a Related Dimension

The `LIMIT.SORTREL` option controls whether or not a sort is done when you limit a dimension to a related dimension. You can suppress the sort that occurs when you limit a dimension to a related dimension by setting `LIMIT.SORTREL` to `no`. This can significantly improve performance when the dimension you are limiting is large.

Note: When `LIMIT.SORTREL` is `NO`, printed output of a dimension may not appear in logical order.

Limiting Based on the Position of a Value in a Dimension

Using the `LIMIT` command, you can set dimension status based on the position of values in either:

- The dimension you are limiting
- An unrelated dimension

Limiting Using Value Position in its Dimension

You can use the `LIMIT` command with the `FIRST`, `LAST`, `NTH`, and `POSLIST` keywords to set dimension status based on the position of a value within a dimension.

The simplified syntax for using the `LIMIT` command in this way is shown below.

```
LIMIT dimension TO {FIRST n|LAST n|NTH n|POSLIST poslist-exp}
```

The FIRST, LAST, and NTH keywords specify where the value is in the full set of dimension values. The *n* argument following it specifies the number of values.

The POSLIST keyword indicates that the *poslist-exp* argument following it is a text expression, each line of which is a numeric value that evaluates to a numeric position of the dimension being limited.

Limiting Using Value Position in an Unrelated Dimension

You can use the LIMIT command with the NOCONVERT keyword to insert a value into a dimension status list based on the numeric position of the values in the status list of the unrelated dimension. This is particularly useful when the two dimensions are in different analytic workspaces (for example, when there is a one-to-one correspondence between the product dimension in two analytic workspaces).

The simplified syntax for using the LIMIT command in this way is shown below:

```
LIMIT dimension TO NOCONVERT unrelated-dimension
```

The *unrelated-dimension* argument specifies the name of a dimension not related to the dimension being limited.

Limiting Based on a Relationship Within a Hierarchy

You can use a family tree to place dimension values in status. You can limit a dimension as follows:

- You can limit a dimension to the parents, children, ancestors, or descendants of each value in a list of specified values or for each value in status.
- You can also find the descendants based on a particular parent relationship. This is useful with hierarchical dimensions that contain both a detail level and levels that are aggregations of lower levels. To use the LIMIT command in this way, you must ensure that the analytic workspace contains a relation that holds the parent for each value of the dimension.

The simplified syntax for limiting a dimension based on a relationship within a hierarchy is shown below.

```
LIMIT dimension TO {PARENTS|CHILDREN|ANCESTORS|DESCENDANTS|HIERARCHY} -
 USING parent-rel[valuelist]
```

The PARENTS keyword finds the parent of each value in *valuelist* or, when there is no *valuelist*, it finds the parent for each value in status. It uses the *parent-rel* to look up the parent.

The `CHILDREN` keyword finds the children of each value in *`valuelist`* or, when there is no *`valuelist`*, finds the children for each value in status. It uses the *`parent-rel`* to look up the children.

The `ANCESTORS` keyword finds the ancestors (that is, parents, grandparents, and so on) of each value in *`valuelist`* or, when there is no *`valuelist`*, finds the ancestors of each value in status.

The `DESCENDANTS` keyword finds the descendants (that is, children, grandchildren, and so on) of each value in *`valuelist`* or, when there is no *`valuelist`*, finds descendants for each value in status.

The `HIERARCHY` keyword is similar to `DESCENDANTS` and finds the descendants (that is, children, grandchildren, and so on) based on the value of the *`parent-rel`* argument.

The *`parent-rel`* argument is the name of a relation between the dimension and itself. For each dimension value, the relation holds another value of the dimension that is its parent dimension value (the one immediately above it in a given hierarchy). This parent-relation can have more than one dimension.

The *`valuelist`* argument can be any inclusive list of values.

See Also:

- "Defining Hierarchical Dimensions and Variables That Use Them" on page 3-22 for more information about hierarchical dimensions.
- "Defining Concat Dimensions and Variables That Use Them" on page on page 3-25 for more information about concat dimensions and hierarchies.
- "Differences Between HIERARCHY and DESCENDANTS Keywords" on page 6-14 for more information about using the HIERARCHY keyword.

Differences Between HIERARCHY and DESCENDANTS Keywords

Both the `HIERARCHY` and `DESCENDANTS` keywords of the `LIMIT` command allow you to set the status of a dimension based on its family tree; however, the different keywords give you different results.

One difference is the order of the values:

- `DESCENDANTS` groups the values by level (all children, and then all grandchildren).
- `HIERARCHY` places each group of children next to its parent.

Additionally, if you use the `HIERARCHY` keyword, then you can include the additional arguments described in the following table that let you further manipulate the contents of the current status list.

IF you want to . . .	**THEN use the . . .**
list children before their parents,	`INVERTED` keyword.
skip *n* generations for each value in *valuelist*, or, when there is no *valuelist* skip *n* generations for each value in status,	`SKIP` *n* phrase.
include *n* generations down from each value of *valuelist* or, when there is no *valuelist*, include *n* generations for each value in status,	`DEPTH` *n* phrase.
run a command, represented as a text expression, every time it constructs a group of children,	`RUN` *textexp* phrase.
exclude the original values from the current status list,	`NOORIGIN` keyword.

Example 6–5 Skipping Generations

Suppose your application issues the following command.

```
LIMIT market TO HIERARCHY DEPTH 2 SKIP 1 USING market.market 'TOTUS'
```

In processing this command, the parent relation is searched (`market.market`) to find the children and the grandchildren (`DEPTH` 2) of `TOTUS` and discards the first generation (`SKIP` 1).

The resulting status follows.

```
TOTUS
BOSTON
ATLANTA
CHICAGO
DALLAS
```

```
DENVER
SEATTLE
```

Note that TOTUS is included in status. With HIERARCHY, the original values are included in status.

Example 6–6 Sorting a Group of Children

When you are using the HIERARCHY keyword with the LIMIT command, you can use the RUN keyword to execute a command, specified as a text expression, every time a group of children is constructed. This lets you further manipulate the values that are being placed in status.

The following command not only limits the values of the market dimension to descendants using the market.market self-relation but also, every time a group of children is constructed, sorts the values in the market dimension in increasing order based on unit sales.

```
LIMIT market TO HIERARCHY RUN 'SORT market A unit.m' USING market.market
```

Note: In this command, if you use KEEP or REMOVE instead of TO in the LIMIT command, then the SORT command has no effect.

Example 6–7 Drilling Down on a Hierarchy Using a Relation

Suppose you want to drill down on districts from the region level of the market dimension. This is a two step process.

The first step in the process is to limit the market dimension, which has embedded totals at the district, region, and total U.S. level, to the region-level data. This is done using the relation mlv.market, which is a relation between market and marketlevel.

The following command produces the report shown below it, which shows the values of mlv.market.

```
REPORT mlv.market

MARKET         MLV.MARKET
-------------- ----------
TOTUS          TOTUS
EAST           REGION
BOSTON         DISTRICT
```

```
ATLANTA        DISTRICT
CENTRAL        REGION
CHICAGO        DISTRICT
DALLAS         DISTRICT
WEST           REGION
DENVER         DISTRICT
SEATTLE        DISTRICT
```

The following commands limit the values of `market` to the desired values and display the values that are currently in status for the `market` dimension.

```
LIMIT market TO mlv.market 'REGION'
STATUS market

The current status of MARKET is:
EAST, CENTRAL, WEST
```

The second step in the process is to drill down on the district-level data from the region level. You can use the self-relation `market.market` to perform the drill down. For each value of the `market` dimension, this relation contains the name of its parent.

```
DEFINE MARKET.MARKET RELATION MARKET <MARKET>
LD Self-relation for the Market Dimension
```

A report of `market.market` produces the following output.

```
MARKET         MARKET.MARKET
-------------- -------------
TOTUS          NA
EAST           TOTUS
BOSTON         CENTRAL
ATLANTA        EAST
CENTRAL        TOTUS
CHICAGO        CENTRAL
DALLAS         CENTRAL
WEST           TOTUS
DENVER         WEST
SEATTLE        WEST
```

The following commands limit `market` to the children of the `EAST`, `CENTRAL`, and `WEST` regions and drill down to the district-level data by using the `CHILDREN` keyword with the `LIMIT` command.

```
LIMIT market TO mlv.market 'REGION'
LIMIT market tO CHILDREN USING market.market
```

A report of `market` produces the following output and shows the values that are now in status.

```
MARKET
-------------
BOSTON
ATLANTA
CHICAGO
DALLAS
DENVER
SEATTLE
```

Limiting Composites and Conjoint Dimensions

You cannot explicitly limit the values of a composite. Composites are not dimensions and, therefore, do not have any independent status. The values of a composite that are in status are determined by the values that are in status in the base dimensions of the composite. In general, when OLAP DML functions and commands deal with objects that are defined with composites, the default behavior is to treat those objects as if no `SPARSE` keyword or named composite had been used when the object was defined.

You can use the `LIMIT` command to set status for the dimensions of a variable that is defined with a composite in the same way you would when the variable is not defined with a composite.

> **See Also:** "Defining Variables That Handle Sparse Data Efficiently" on page 3-18 for more information about composites.

Example 6–8 Limiting Dimensions Used by a Composite

Suppose your analytic workspace contains a variable named `coupons` that is dimensioned by `month` and (using the `prod_market` composite) `product` and `market` as shown in the following definition.

```
DEFINE coupons VARIABLE INTEGER <month prod_market <product market>>
```

The following commands display the default status of all of the base dimensions of the `coupons` variable.

```
STATUS coupons

The current status of MONTH is:
ALL
The current status of PRODUCT is:
ALL
The current status of MARKET is:
ALL
```

Later, when you want to access only the values of `coupon` that apply to sportswear, you limit the base dimension `product` as shown below.

```
LIMIT product TO 'SPORTSWEAR'
```

Ways of Limiting Conjoint Dimensions

You can limit a conjoint dimension in either of the following ways:

- Limit the base dimensions.
- Limit the conjoint dimension itself.

See Also: "Defining Variables That Handle Sparse Data Efficiently" on page 3-18 for more information about conjoint dimensions.

Limiting Conjoint Dimensions Using Value Combinations

To limit a conjoint dimension to a list of values, you can use the following constructions:

- Specify the actual values, surrounding each combination with angle brackets.

  ```
  LIMIT proddist TO <'TENTS' 'BOSTON'> <'FOOTWEAR' 'DENVER'>
  ```

- Use a variable name for the values, surrounding the combination with angle brackets.

  ```
  prodname = 'CANOES'
  distname = 'SEATTLE'
  LIMIT proddist TO <prodname distname>
  ```

- Create a multiline list, in which each line is a combination surrounded by angle brackets and separated by `\n` (the linefeed escape sequence).

```
namelist = mytext = '<\'TENTS\' \'BOSTON\'>\n <\'FOOTWEAR\' \'DENVER\'>'
LIMIT proddist TO namelist
```

Limiting Conjoint Dimensions Using Base Dimension Values

Because there is an implicit relation between a conjoint dimension and its base dimensions, you can limit the conjoint dimension by limiting the base dimensions.

For example, the following command limits a conjoint dimension named `proddist` to all conjoint values having `CANOES` as one of the values of the base dimension `product`.

```
LIMIT proddist TO product 'CANOES'
```

Limiting Concat Dimensions

The current status list of a concat dimension is separate from the current status lists of its base dimensions. However, you limit a concat dimension by specifying values of its base dimensions.

In Example 6–9, the base dimensions of the concat dimension `reg.dist.cc` are the simple dimension `region` and the conjoint dimension `proddist`. The example limits the concat dimension to the `WEST` region and `proddist` to the conjoint values `TENTS DENVER` and `RACQUETS DENVER` and then reports the values of the concat dimension.

Example 6–9 Limiting Base Dimensions of a Concat Dimension

```
LIMIT reg.dist.ccdim TO region'WEST'
LIMIT reg.proddist.ccdim ADD proddist <'TENTS' 'DENVER'> -
<'RACQUETS', 'DENVER'>

REPORT reg.proddist.ccdim

REG.PRODDIST.CCDIM
------------------------------
<REGION: WEST>
<PRODDIST: <TENTS, DENVER>>
<PRODDIST: <RACQUETS, DENVER>>
```

Working with Null Status

You can set the current status list of a dimension to null (empty status) only when you have explicitly specified that you want null status to be permitted. You can give this permission in either of two ways:

- Set the `OKNULLSTATUS` option to `yes`. This specification indicates that null status should be allowed whenever it occurs except when the `IFNONE` argument is present in a `LIMIT` command.
- Use the `NULL` keyword in a `LIMIT` command to set the status of a particular dimension or valueset to null. You can do this by specifying `TO NULL` or `KEEP NULL`. This specification indicates that null status should be allowed for this `LIMIT` command only.

If you have not used either of these two methods to give permission for null status and you execute a `LIMIT` command that would result in null status, then the status is not changed to null when the command is executed. Instead, the status remains the same as it was before the command was issued.

You cannot use the `IFNONE` and `NULL` keywords in the same `LIMIT` command.

Managing Null Status in a Program

An `IFNONE` argument in a `LIMIT` command indicates that you do not want program execution to take its normal course when a dimension status is set to null. Therefore, when `IFNONE` is present, a branch is performed to the IFNONE label and the status is not set to null, even if `OKNULLSTATUS` is `YES`. If the `NULL` keyword is present together with `IFNONE`, then the inconsistency is signaled with an error.

Tip: Using the `IFNONE` argument provides limited flexibility for handling null status because it simply branches to a label. For more flexibility, investigate the possibility of setting the `OKNULLSTATUS` option to control whether or not execution will branch when status is null, and the possibility of using a `WHILE` loop to test for null status.

Errors When Limiting Status to a Null Value

An error will not be signaled when you try to limit the status of a dimension or valueset that has no values, unless you explicitly list values that do not exist. For

example, if you have not added any values to a newly defined dimension WEEK, then the following command does not cause an error.

```
LIMIT week TO FIRST 10
```

However, the following command does cause an error because PETE is not a value.

```
LIMIT week TO 'PETE'
```

Similarly, the following command causes an error because WEEK does not have a value at position 20.

```
LIMIT week TO 20
```

Working with Valuesets

A valueset is a workspace object that contains a list of dimension values for a particular dimension. You use a valueset to save a dimension status list for later use. The values in a valueset can be saved across OLAP sessions. When you attach an analytic workspace, each dimension has all of the values in the default status list. You can then limit a dimension to the values stored in the valueset for that dimension. When you first define a valueset, its value is null. After defining a valueset, you use the LIMIT command to assign values from the dimension to the valueset. You can use the LIMIT command with valuesets in many of the ways that you use it with dimensions. For example, you can use the LIMIT command to expand, reduce, and replace values in the list of values of a valueset.

Creating a Valueset

To create a valueset, take the following steps.

1. Define a valueset for the dimension values. Use the DEFINE command with the VALUESET keyword.
2. Limit the dimension for which you want to create a valueset to the values you want to save.
3. Limit the valueset you created in Step 1 to the dimension you limited in Step 2.

Example 6–10 Creating a Valueset

This example defines a valueset named lineset. It is dimensioned by line and, therefore, it can be limited by the current values of the line dimension.

The following commands limit the `line` dimension to the first two values, then show the current status of `line`.

```
LIMIT line TO FIRST 2
STATUS line

The current status of LINE is:
REVENUE, COGS
```

These commands define a valueset names `lineset`, set it to the current status list of the `line` dimension, and show its values. The `LD` command attaches a description to the object.

```
DEFINE lineset VALUESET line
LD Valueset for LINE dimension values
LIMIT lineset TO line
SHOW VALUES(lineset)

REVENUE
COGS
```

Limiting Using a Valueset

After you have defined a valueset, you can use it to limit a dimension with a single `LIMIT` command.

For example, the following commands limit the `line` dimension to the values stored in the `lineset` valueset and display the new status of `line`.

```
LIMIT line TO lineset
STATUS line

The current status of LINE is:
REVENUE, COGS
```

Example 6–11 Limiting Using a Valueset

The following commands limit `district` to the districts in which sportswear sales exceeded $1,000,000 in 1996. The current status list for the `district` dimension is saved in the valueset `SPORTS.DISTRICT`. Once you have created the valueset, you can limit the `district` dimension to the same values with one `LIMIT` command.

```
DEFINE sports.district VALUESET district
LIMIT product TO 'SPORTSWEAR'
LIMIT month TO year 'YR96'
```

```
LIMIT sports.district TO TOTAL(sales district) GT 1000000
LIMIT district TO sports.district
```

The `STATUS` command shows the new status of `district`.

```
STATUS district

The current status of DISTRICT is:
ATLANTA TO DENVER
```

Changing the Values of a Valueset

You can use the `LIMIT` command to change the values in a valueset. The simplified syntax for using the `LIMIT` command in this way is shown below:

```
LIMIT valueset keyword selection
```

The *valueset* argument specifies the name of the valueset you want to change.

The *keyword* that you specify determines how the command affects the values that are currently in the valueset. The following table outlines the use of the keywords.

IF you want to . . .	THEN use the LIMIT command with . . .
replace the values that are currently in the valueset with new values,	either the `TO` or `COMPLEMENT` keyword.
remove values from the current valueset,	either the `REMOVE` or `KEEP` keyword.
expand the valueset,	either the `ADD` or `INSERT` keyword.
sort the values in the valueset,	the `SORT` keyword.

The *selection* argument specifies the selection criteria that you want to be used to determine what values to assign to the valueset. In general, you can use the same arguments when you are using the `LIMIT` command to select values for a valueset that you can use when you use the `LIMIT` command to limit a dimension.

Identifying and Retrieving the Values in a Valueset

You can use the following commands and functions to identify and retrieve dimension values that are in a valueset.

Command or function	Description
`INSTAT` function	Checks whether a dimension value is in a valueset.
`STATFIRST` function	Retrieves the first value in a valueset.
`STATLAST` function	Retrieves the last value in a valueset.
`STATUS` command	Sends to the current outfile the status of one or more values in a valueset.
`VALUES` function	Retrieves the values in a valueset. Depending on whether you specify the `INTEGER` keyword, the function either returns a multiline text value that contains one dimension value per line or returns, as integers, the position numbers of the values in the existing dimension, not in the valueset.

Retrieving the Values in a Valueset

Suppose an analytic workspace contains a valueset called `monthset` that has the values `JAN95`, `MAY95`, and `DEC95`. You can use the `VALUES` function to list the values in that valueset.

The following OLAP DML command produces the output shown below it.

```
SHOW VALUES(monthset)

JAN95
MAY95
DEC95
```

Retrieving the Dimension Positions of Values in a Valueset

Suppose that you want to retrieve the position of the values in the `monthset` valueset, rather than retrieve the actual values themselves. To retrieve the position of values, you use the `VALUES` function with the `INTEGER` keyword. When you use this keyword, the position numbers are returned instead of the actual dimension values that are included in a valueset. The position numbers that are returned do not represent positions in the valueset; they represent positions in the dimension on which the valueset is based.

The following OLAP DML command produces the output shown below it.

```
SHOW VALUES(monthset INTEGER)

61
65
72
```

The value JAN95 is shown as the sixty-first value in the month dimension, MAY95 as the sixty-fifth value, and DEC95 as the seventy-second value, although they are the first, second, and third values in monthset.

Part II

Applications Development

Part II contains information of particular interest to applications developers.

It contains the following chapters:

- Chapter 7, "Developing Programs"
- Chapter 8, "Working with Models"
- Chapter 9, "Allocating Data"

7 Developing Programs

This chapter provides information about writing, compiling, testing, and calling programs that are written in the OLAP DML. It includes the following topics:

- Introduction to OLAP DML Programs
- Defining and Editing Programs
- Using Variables in Programs
- Passing Arguments
- Writing User-Defined Functions
- Controlling the Flow of Execution
- Directing Output
- Preserving the Session Environment
- Handling Errors
- Compiling Programs
- Testing and Debugging Programs

Introduction to OLAP DML Programs

An OLAP DML program is written in the OLAP DML. It acts on data in the analytic workspace and helps you accomplish some workspace management or analysis task. You can write OLAP DML programs to perform tasks that you must do repeatedly in the analytic workspace, or you can write them as part of an application that you are developing.

There are two main types of OLAP DML programs: programs that do *not* return values, and programs that return values. A program that returns a value is called a user-defined function.

You can use an OLAP DML program that does not return a value as a standalone program or as the main program or subprogram of a multiprogram application. These programs behave like OLAP DML commands.

You can use a user-defined function in commands and expressions in the same way that you use built-in OLAP DML functions.

In contrast to the form of a program, the content is related to the job it was created to do, and it is the individual lines of a program that provide its content. Program lines that accomplish specific purposes are discussed in other chapters in this guide.

Executing Programs

You can invoke a program that does not return a value by using the `CALL` command. You enclose arguments in parentheses, and they are passed by value.

For example, suppose you create a simple program named `addit` to add two integers. You can use the `CALL` command in the main program of your application to invoke the program.

```
CALL addit (3, 4)
```

The syntax for using the `CALL` command to invoke a program is shown below.

```
CALL program-name [(arg1 [arg2 ...])]
```

The *`program-name`* argument is the name of the program to be called.

The *`arg1...`* arguments are optional and specify any arguments that are expected by the called program. Specify the arguments so that they match the order in which they are defined in the program.

Executing User-Defined Functions

A user-defined function is a program that returns a value. You invoke user-defined functions in the same way as you use built-in functions. You merely use the program's name in an expression and enclose the program's arguments, if any, in parentheses.

For example:

- You can use the program name as an expression in a command.

  The following `REPORT` command uses the value that is returned by the user-defined function `isrecent` that has a single argument, `actual`.

  ```
  REPORT isrecent(actual)
  ```

- You can use the = command to assign the return value of the function to a variable.

  The following command assigns the return value of the user-defined function named `tempsales` to a temporary variable called `mytempsales`.

  ```
  mytempsales = tempsales
  ```

Important: Although you can also run user-defined functions using the `CALL` command, you will not be able to access the return value.

Defining and Editing Programs

A program, like a dimension or a variable, is a workspace object. You define a program using the `DEFINE` command. The following example defines a program named `hello`.

```
DEFINE hello PROGRAM
```

Once you have defined a program object, you need to add the body of the program to it.

OLAP Worksheet provides an editor that you can use to add content to the program definition.

> **See Also:** "Accessing a Workspace from OLAP Worksheet" on page 1-6 for more information about using OLAP Worksheet.

Formatting Guidelines for Editing Programs

Use the following formatting guidelines as you add lines to your program:

- Each line of code can have a maximum of 4000 bytes.
- To continue a single command on the next line, place a hyphen (-) at the end of the line to be broken. The hyphen is called a continuation character.

  You cannot use a continuation character in the middle of a text literal.
- To write more than one command on a single line, separate the commands with semicolon (;).
- Enclose literal text in single quotation marks ('). To include a single quotation mark within literal text, precede it with a backslash (\).
- Precede comments with double quotation marks ("). You can place a comment, preceded by double quotation marks, either at the beginning of a line or at the end of a line, after some commands.

The following program named `hello` displays the phrase “Hello World.”

```
DEFINE hello PROGRAM
PROGRAM
SHOW 'Hello World'
END
```

> **See Also:** "Escape Sequences" on page 3-6 for information about escape sequences.

Using Variables in Programs

Variables that hold the data in your analytic workspaces are permanent variables. These variables persist from one OLAP session to another. However, you might not need to save variables that your programs use to hold processing information while

they manipulate data. So that you do not clutter your analytic workspaces with unnecessary variables, you can define temporary and local variables:

- A temporary variable has a value only during the current session. When you update and commit the analytic workspace, only the definitions of the variables are saved. When you detach the analytic workspace, the data values are discarded.
- A local variable is a single-cell variable that exists only while the program in which it is defined is running. Using local variables within a program is a useful alternative to using temporary variables.

  Local variables have no dimensions, so you cannot use them for storing dimensioned data. Because they exist only for the duration of the program in which they are defined, you cannot store information in a local variable in one program and then use that variable in another program. If you must store dimensioned data, or use information in more than one program, then define a temporary variable instead.

Global Versus Modular Design Approaches

The purpose of most OLAP DML programs is to manipulate data. Depending on your programming style and the requirements of your application, you might use either of the following approaches:

- Use permanent variables, to which all programs have access. This approach requires less programming overhead (for example, fewer definitions), but it is less modular. If you are not careful, then programs can interfere with one another when they set the values of permanent variables.
- Use program arguments, local variables, and return values from user-defined functions. This approach forces you to write modular programs with clear input and output responsibilities.

Most applications combine these approaches, using permanent variables and user-defined functions when they are appropriate. In general, modular programs are considered to be easier to read, debug, and maintain.

Defining Temporary Variables

You define temporary variables with the `TEMP` keyword in the `DEFINE` command, as in the following example.

```
DEFINE total.sales DECIMAL TEMP
```

Defining temporary variables for use in programs helps you avoid cluttering your analytic workspace with temporary data, but it still adds objects to your analytic workspace. For most simple applications, the addition of a few temporary objects is not a problem. However, in complex applications that require many programs, the number of temporary objects can sometimes get very large, and this can affect the application's performance.

Once defined, a temporary variable will exist for the remainder of a user's session unless it is deleted. Be sure to delete the temporary variable as part of the cleanup of your program, or create it on the condition that it does not already exist, so that it can be rerun during a session without causing an error.

Defining Local Variables

You must specify local variables at the beginning of your program, before any executable commands. You specify a local variable with the `VARIABLE` command, which has the following syntax.

```
VARIABLE name datatype
```

The *name* argument specifies the name of the variable. To minimize confusion or problems, you should avoid using the same name for both an analytic workspace variable and a local variable. When both an analytic workspace variable and a local variable have the same name, then the local variable usually takes precedence. However, in a few commands and functions that operate on workspace objects (for example, the OBJ function), the defined variable takes precedence.

The *datatype* argument specifies the data type of the local variable. For more information on data types, see "Data Types" on page 3-4.

The program named `west.rpt`, listed below, includes definitions for two local variables named `data` and `rpt.month`.

```
DEFINE west.rpt PROGRAM
LD Produce report for Western Sales District
PROGRAM
VARIABLE data TEXT
VARIABLE rpt.month TEXT
LIMIT month TO LAST 3
   .
   .
   .
```

Passing Arguments

The OLAP DML provides two ways for you to accept arguments in a program:

- ARGUMENT command. You can use the ARGUMENT command to declare arguments in a program. ARGUMENT command allows you to use both simple and complex arguments (such as expressions). The ARGUMENT command also makes it convenient to pass arguments from one program to another, or to create your own user-defined functions.
- ARG functions. You can use the ARG, ARGS, and ARGFR functions in any program to retrieve arguments from a command. These functions are primarily useful for simple text arguments.

Using the ARGUMENT Command

The ARGUMENT command lets you declare an argument of any data type, dimension, or valueset. Any ARGUMENT commands must precede the first executable line in the program. When you run the program, these declared arguments are initialized with the values you provided as arguments to the program. The program can then use these arguments in the same way it would use local variables.

Example 7–1 Using the ARGUMENT Command

Suppose you are writing a program, called product.rpt. The product.rpt program produces a report, and you want to supply an argument to the report program that specifies the text that should appear for an NA value in the report. In the product.rpt program, you can use the declared argument natext in an = command to set the NASPELL option to the value provided as an argument.

```
ARGUMENT natext TEXT
NASPELL = natext
```

To specify Missing as the text for NA values, you can execute the following command.

```
CALL product.rpt ('Missing')
```

In this example, literal text enclosed in single quotes provides the value of the text argument. However, any other type of text expression works equally well, as shown in the next example.

```
DEFINE natemp VARIABLE TEXT TEMP
natemp = 'Missing'
CALL product.rpt (natemp)
```

Using Multiple Arguments

A program can declare as many arguments as needed. When the program is executed with arguments specified, the arguments are matched positionally with the declared arguments in the program.

When you run the program, you must separate arguments with spaces rather than with commas or other punctuation. Punctuation is treated as part of the arguments.

Example 7–2 Passing Multiple Arguments

Suppose, in the `product.rpt` program, that you want to supply a second argument that specifies the column width for the data columns in the report. In the `product.rpt` program, you would add a second `ARGUMENT` command to declare the integer argument to be used in setting the value of the `COLWIDTH` option.

```
ARGUMENT natext TEXT
ARGUMENT widthamt INTEGER
NASPELL = natext
COLWIDTH = widthamt
```

To specify eight-character columns, you could run the `product.rpt` program with the following command.

```
CALL product.rpt ('Missing' 8)
```

If the `product.rpt` program also requires the name of a product as a third argument, then in the `product.rpt` program you would add a third `ARGUMENT` command to handle the product argument, and you would set the status of the `product` dimension using this argument.

```
ARGUMENT natext TEXT
ARGUMENT widthamt INTEGER
ARGUMENT rptprod PRODUCT
NASPELL = natext
COLWIDTH = widthamt
LIMIT product TO rptprod
```

You can run the `product.rpt` program with the following command.

```
CALL product.rpt ('Missing' 8 'TENTS')
```

In this example, the third argument is specified in uppercase letters with the assumption that all the dimension values in the analytic workspace are in uppercase letters.

Passing Arguments as Text with Ampersand Substitution

It is very common to pass a simple text argument to a program. However, there are some situations in which you might want to pass a more complicated text argument, such as an argument that is composed of more than one dimension value or is composed of the text of an expression. In these cases, you want to substitute the text you pass, exactly as you specify it, wherever the argument name appears.

To indicate that you want a text argument handled in this way, you precede the argument name with an ampersand when you use it in the command lines of your program. Specifying arguments in this way is called **ampersand substitution**.

When you use ampersand substitution to pass the names of workspace objects to a program (rather than their values), the program has access to the objects themselves because the names are known to the program. This is useful when the program must manipulate the objects in several operations.

Important: You cannot compile and save any program line that contains an ampersand. Instead, the line is evaluated at run time, which can reduce the speed of your programs. Therefore, to maximize performance, avoid using ampersand substitution when another technique is available.

Example 7–3 Passing Multiple Dimension Values

If you want to specify exactly two products for the `product.rpt` program discussed earlier, then you could declare two dimension-value arguments to handle them. But if you want to be able to specify any number of products using `LIMIT` keywords, then you can use a single argument with ampersand substitution.

Suppose you use the following commands in your program.

```
ARGUMENT natext TEXT
ARGUMENT widthamt INTEGER
ARGUMENT rptprod TEXT
     .
     .
     .
LIMIT product TO &rptprod
```

You can run the program and specify that you want the first three products in the report.

```
CALL product.rpt ('Missing' 8 'first 3')
```

The single quotation marks are necessary to indicate that "first 3" should be taken as a single argument, rather than two separate arguments separated by a space. The ampersand causes `LIMIT` to interpret `'first 3'` as a keyword expression rather than as a dimension value.

Example 7–4 Passing the Text of an Expression

Suppose you have a program named `custom.rpt` that includes a `REPORT` command, but you want to be able to use the program to present the values of an expression, such as `sales - expense`, as well as single variables.

```
custom.rpt 'sales - expense'
```

> **Note:** You must enclose the expression in single quotation marks. Because the expression contains punctuation (the minus sign), the quotation marks are necessary to indicate that the entire expression is a single argument.

In the `custom.rpt` program, you could use the following commands to produce a report of this expression.

```
ARGUMENT rptexp TEXT
REPORT &rptexp
```

Passing Object Names and Keywords

For the following types of arguments, you must always use an ampersand to make the appropriate substitution:

- Names of workspace objects, such as `units` or `product`
- Command keywords, such as `COMMA` or `NOCOMMA` in the `REPORT` command, or `A` or `D` in the `SORT` command

Example 7–5 Passing Workspace Object Names and Keywords

Suppose you design a program called `sales.rpt` that produces a report on a variable that is specified as an argument and sorts the `product` dimension in the order that is specified in another argument. You would run the `sales.rpt` program by executing a command like the following one.

```
sales.rpt units d
```

In the `sales.rpt` program, you can use the following commands.

```
ARGUMENT varname TEXT
ARGUMENT sortkey TEXT
SORT product &sortkey &varname
REPORT &varname
```

After substituting the arguments, these commands are executed in the `sales.rpt` program.

```
SORT product D units
REPORT units
```

See Also: "Substitution Expressions" on page 4-30 for more information about ampersand substitution.

Writing User-Defined Functions

When an OLAP DML program returns a value, it is called a user-defined function. You can use it in commands and expressions.

A user-defined function contains a `RETURN` command followed by an expression.

```
RETURN expression
```

The `RETURN` command returns a single value when the program terminates.

Data Type of a User-Defined Function

When you create a user-defined function, you define the program with a data type or dimension name, using the following syntax of the DEFINE command.

```
DEFINE programname PROGRAM [datatype|dimension]
```

The *datatype* argument specifies the data type of the value to be returned by the program when it is called as a function.

The *dimension* argument specifies the name of a dimension whose value the program returns when it is called as a function. The return value will be a single value of the dimension, not a position (integer). The dimension must be defined in the same analytic workspace as the program. The value that is returned by the program has the data type that is specified in the definition. If you specify a dimension name, then the program returns a value of that dimension.

The return expression in the program should match the data type that is specified in its definition. If the data type of the return value does not match the data type that is specified in its definition, then the value is converted to the data type in the definition.

If you do not specify a data type for the program, then the return value is converted to the data type that is required by the caller.

Arguments in a User-Defined Function

User-defined functions can accept arguments. A user-defined function returns only a single value. However, if you supply an argument to a user-defined function in a context that loops over a dimension (for example, in a REPORT command), then the function returns results with the same dimensions as its argument.

You must declare the arguments using the ARGUMENT command within the program, and you must specify the arguments in parentheses following the name of the program.

> **See Also:** "Passing Arguments" on page 7-7 for more information about using arguments with programs.

Example 7–6 User-Defined Function

Suppose your analytic workspace contains a variable called units.plan, which is dimensioned by the product, district, and month dimensions. The variable

holds integer data that indicates the number of product units that are expected to be sold.

Suppose also that you define a program named `units_goals_met`. This program is a user-defined function. It accepts three dimension-value arguments that specify a given cell of the `units.plan` variable, and it accepts a fourth argument that specifies the number of units that were actually sold for that cell. The program returns a Boolean value to the calling program. It returns `YES` when the actual figure comes up to within 10 percent of the planned figure; it returns `NO` when the actual figure does not.

The definition of the `units_goals_met` program is listed below.

```
DEFINE units_goal_met PROGRAM BOOLEAN
LD Tests whether actual units met the planned estimate
"Program Initialization
ARGUMENT userprod  TEXT
ARGUMENT userdist  TEXT
ARGUMENT usermonth TEXT
ARGUMENT userunits integer
VARIABLE answer boolean
TRAP ON errorlabel
PUSH product district month
"Program Body
LIMIT product TO userprod
LIMIT district TO userdist
LIMIT month TO usermonth
IF (units.plan - userunits) / units.plan GT .10
   THEN answer = NO
   ELSE answer = YES
"Normal Exit
POP product district month
RETURN answer
"Abnormal Exit
errorlabel:
POP product district month
SIGNAL errorname errortext
END
```

To execute the `units_goal_met` program and store the return value in a variable called `success`, you can use an assignment statement.

```
success = units_goal_met('TENTS' 'BOSTON' 'JUN96' 2000)
```

Controlling the Flow of Execution

Ordinarily, the lines of a program are executed sequentially, that is, in linear fashion. However, a well-designed program controls the flow of execution by using commands that redirect the path of execution when appropriate.

You can use the following control structures to modify the sequence of command execution.

Command or Keyword	Action	Event that Triggers Action
IF command	Executes alternative commands or groups of commands.	A specified Boolean condition is or is not met.
WHILE command	Executes a group of commands repeatedly.	As long as a specified Boolean condition is met.
FOR command	Executes a command or a group of commands.	Once for each value of a dimension.
GOTO command	Branches to a specific labeled location.	Issuing the command.
SWITCH command	Branches to particular branch in a multipath branch.	A specific criterion is met.
TRAP command	Branches to a specific labeled location.	An error occurs during program execution.
IFNONE keyword in a LIMIT, REPORT, ROW, or HEADING command	Branches to a specific labeled location.	An attempt to set status would result in no values or null status.
RETURN command	Branches out of a program or returns to a calling program before the final command in the program.	Issuing the command.

Guidelines for Constructing a Label

When you use control structures to branch to a particular location, you must provide a label for the location in order to identify it clearly. When creating a label, follow these guidelines:

- The first character in the label must be a letter, period (.), or underscore (_).
- The remaining characters in a label can be any combination of letters, numbers, periods, or underscores.

- A label must be followed immediately by a colon (`:`).
- Make sure that the first eight bytes in the label are unique. (Note that, in your character set, a byte might or might not be equivalent to one character.) A label can contain up to 3999 bytes (the maximum length of a text line minus 1 byte for the colon that identifies a label). However, because only the first eight bytes of a label name are used, you can experience problems with label names greater than eight bytes when the first eight bytes are not unique.

Alternatives to the GOTO Command

While `GOTO` makes it easy to branch within a program, frequent use of it can obscure the logic of your program, making it difficult to follow its flow. This is particularly true when you have a complex program with several labels and `GOTO` commands that skip over large portions of code.

To keep the logic of your programs clear, minimize your use of `GOTO`.

Sometimes a `GOTO` command is the best programming technique, but often there are better alternatives. For example:

- Instead of using `GOTO` commands in an `IF` command, you can often place your alternative sets of commands between `DO` and `DOEND` commands within the `IF` command itself.
- If each set of commands is long or you want to use them in more than one place in your program, then you might consider placing them in subprograms. Then, you can use the `IF` command to choose between two different programs, or use the `SWITCH` command to choose among many different programs.

Example 7–7 Using the FOR Command for Looping Over Dimension Values

The `FOR` command executes the commands in the loop for each value in the current status of the dimension. You must limit the dimension to the desired values before executing the `FOR` command. For example, you can produce a series of output lines that show the price for each product.

```
LIMIT month TO FIRST 1
LIMIT product TO ALL
FOR product
SHOW JOINCHARS('Price for ' product ': $' price)
```

Each output line has the following format.

```
Price for TENTS: $165.50
```

When your data is multidimensional, you can specify more than one dimension in a FOR command to control the order of processing. For example, you can use the following command to control the order in which dimension values of the units data are processed.

```
FOR month district product
   units = ...
```

When this assignment statement is executed, the month dimension varies the slowest, the district dimension varies the next slowest, and the product dimension varies the fastest. Thus, a loop is performed over all products for the first district before doing the next district, and over all districts for the first month before doing the next month.

Within the FOR loop, each specified dimension is temporarily limited to a single value while it executes the commands in the loop. You can therefore work with specific combinations of dimension values within the loop.

Example 7–8 Using DO/DOEND in a FOR Loop

If actual figures for unit sales are stored in a variable called units and projected figures for unit sales are stored in a variable called units.plan, then the code in your loop can compare these figures for the same combination of dimension values.

```
LIMIT month TO FIRST 1
LIMIT product TO ALL
LIMIT district TO ALL
FOR district product
   DO
     IF (units.plan - units)/units.plan GT .1
     THEN SHOW JOINCHARS(-
       'Unit sales for ' product ' in ' -
       district ' are not within 10% of plan.')
   DOEND
```

These lines of code are processed as described below.

1. The data is limited to a specific month.
2. All the districts and products are placed in status, and the FOR loop is entered.
3. In the FOR loop, the actual figure is tested against the planned figure. If the unit sales figure for TENTS in BOSTON is more than 10 percent below the planned figure, then the following message is sent to the current outfile.

   ```
   Unit sales for TENTS in BOSTON are not within 10% of plan.
   ```

4. After processing all the products, the FOR loop is complete for the first district.
5. The loop is executed for the second district, and so on.

Note: While the FOR loop executes, each dimension that is specified in a FOR command is limited temporarily to a single value. If you specify district in the FOR loop, but not product, then all the values of product are in status while the FOR loop executes. The IF command then tests data for only the first value of the product dimension.

Example 7–9 Branching to Avoid Setting Null Status

Your program might try to set or refine the status of the product dimension to include only the products for which unit sales are greater than 500. If no products have unit sales of more than 500, then you can use the IFNONE keyword to specify that execution branch to the novals label.

```
LIMIT product KEEP units GT 500 IFNONE novals
```

In the commands following the novals label, you can handle the special situation in which no products have units sales greater than 500.

Example 7–10 Branching After Setting Null Status

As an alternative to branching to an IFNONE label, you can also handle null status for a dimension with the OKNULLSTATUS option. If you set OKNULLSTATUS to YES, then you will be allowed to set the status of a dimension to null. You can then check for null status and execute appropriate commands with an IF command, or you can handle null status as one of the cases in a SWITCH command.

```
OKNULLSTATUS = YES
LIMIT month TO sales GT salesnum
IF STATLEN(month) LT 1
   THEN GOTO showerr
```

Directing Output

To send output to a file, use the OUTFILE command followed by a directory alias and a file name, and separatae the two with a slash (/). A file will be created with the name you specify. Before you execute the OUTFILE command, you can use the

CDA command to specify a current directory alias. In this case, you do not have to specify a directory alias in the OUTFILE command because Oracle OLAP assumes that you want the file to be created in your current directory alias.

Directory aliases are defined in the database and control access to directories. Contact your Oracle DBA for the name of a directory alias to which your database user name has read/write access. The file name that you specify must follow the standard filename format for your operating system.

The OUTFILE command changes the routing for all subsequent output. Therefore, if your program routes a report to a file, then you should reroute output to the default outfile before leaving the program. If you want to send subsequent output to the default outfile, then place the OUTFILE EOF command directly after your report commands. To make sure the OUTFILE EOF command is executed when errors cause abnormal termination of the program, also place the command in the abnormal exit section.

If you are working in OLAP Worksheet, the default outfile is its response window. The current destination is called the current outfile.

Example 7–11 Directing Output to a File

Suppose you have a program called year.end.sales, and you want to save the report it creates in a file. Type the following commands to write a file of the report. In this example, userfiles is a directory alias and yearend.txt is the name of the file.

```
OUTFILE 'userfiles/yearend.txt'
year.end.sales
OUTFILE EOF
```

Now the file contains the year.end.sales report. You can add more reports to the same file with the APPEND keyword for OUTFILE. Suppose you have another program called year.end.expenses. Add its report to the file with the following commands. Note that without APPEND, the OUTFILE command overwrites the expense report.

```
OUTFILE APPEND 'userfiles/yearend.txt'
year.end.expenses
OUTFILE EOF
```

Capturing Error Messages

You can capture error messages by setting the ECHOPROMPT option to YES.

```
ECHOPROMPT = YES
```

When you set ECHOPROMPT to YES, input lines and error messages are echoed, as well as output lines, to the current outfile. If you use the OUTFILE or DBGOUTFILE command, you can capture the error messages in a file. For information about DBGOUTFILE, see "Sending Output to a Debugging File" on page 7-31.

Whenever you change a setting, remember to save and restore its original value with the PUSH and POP commands.

Preserving the Session Environment

One advantage to the modular design approach is that each program has a clearly defined area of responsibility, and it does not affect the workings of other programs. To make this possible, each program must act as a "good citizen" by saving global settings before it changes them and restoring global settings before it finishes execution.

There are two types of environment settings:

- Session environment. The dimension status, option values, and output destination that are in effect before a program is run make up the session environment.
- Program environment. The dimension status, option values, and output destination that you use in a program make up the program environment.

Changing the Program Environment

To perform a task within a program, you often need to change the output destination or some dimension and option values. For example, you might run a monthly sales report that always shows the last six months of sales data. You might want to show the data without decimal places, include the text "No Sales" where the sales figure is zero, and send the report to a file. To set up this program environment, you can use the following commands in your program.

```
LIMIT month TO LAST 6
DECIMALS = 0
ZSPELL = 'No Sales'
OUTFILE monsales.txt
```

To avoid disrupting the session environment, the initialization section of a program should save the values of the dimensions and options that will be set in the program. In the normal and abnormal exit sections at the end of the program, you can restore the saved environment, so that other programs do not need to be concerned about whether any values have been changed. In addition, if you have sent output to a file, then the exit sections should return the output destination to the default outfile.

Ways to Save and Restore Environments

The following suggestions let you save the environment of a program or a session:

- If you want to save the current status or value of a dimension, a valueset, an option, or a single-cell variable for use in the current program, then use the `PUSHLEVEL` and `PUSH` commands. You can restore the current status values using the `POPLEVEL` and `POP` commands.
- If you want to save, access, or update the current status or value of a dimension, a valueset, an option, a single-cell variable, or a single-cell relation for use in the current session, then use a named context. Use the `CONTEXT` command to define the context.

Contexts are the most sophisticated way to save object values for use during a session. With contexts, you can access, update, and commit the saved object values. In contrast, `PUSH` and `POP` simply allow you to save and restore values. Typically, you use the `PUSH` and `POP` commands within a program to make changes that apply only during the program's execution.

Saving the Status of a Dimension or the Value of an Option

The `PUSH` command saves the current status of a dimension, the value of an option, or the value of a single-cell variable. For example, to save the current value of the `DECIMALS` option so you can set it to a different value for the duration of the program, use the following command in the initialization section.

```
PUSH DECIMALS
```

You do not need to know the original value of the option to save it or to restore it later. You can restore the saved value with the `POP` command.

```
POP DECIMALS
```

You must make sure the `POP` command is executed when errors cause abnormal termination of the program as well as when the program ends normally. Therefore,

you should place the POP command in the normal and abnormal exit sections of the program.

Saving Several Values at Once

You can save the status of one or more dimensions and the values of any number of options and variables in a single PUSH command, and you can restore the values with a single POP command, as shown in the following example.

```
PUSH month DECIMALS ZSPELL
       .
       .
       .
POP month DECIMALS ZSPELL
```

Using Level Markers

If you are saving the values of several dimensions and options, then the PUSHLEVEL and POPLEVEL commands provide a convenient way to save and restore the session environment.

You first use the PUSHLEVEL command to establish a level marker. Once the level marker is established, you use the PUSH command to save the status of dimensions and the values of options or single-cell variables.

If you place more than one PUSH command between the PUSHLEVEL and POPLEVEL commands, then all the objects that are specified in those PUSH commands are restored with a single POPLEVEL command.

By using PUSHLEVEL and POPLEVEL, you save some typing as you write your program because you only need to type the list of objects once. You also reduce the risk of omitting an object from the list or misspelling the name of an object.

Example 7–12 Creating Level Markers

For example, you can use the PUSHLEVEL command to establish a level marker called firstlevel, and then use PUSH to save the current values.

```
PUSHLEVEL 'firstlevel'
PUSH month DECIMALS ZSPELL
```

The level marker can be any text that is enclosed in single quotation marks. It can also be the name of a single-cell ID or TEXT variable, whose value becomes the name of the level marker. In the exit sections of the program, you can then use the

POPLEVEL command to restore all the values you saved since establishing the firstlevel marker.

```
POPLEVEL 'firstlevel'
```

Example 7–13 Nesting PUSHLEVEL and POPLEVEL Commands

You can nest PUSHLEVEL and POPLEVEL commands to save certain groups of values in one place in a program and other groups of values in another place in a program. The next example shows two sets of nested PUSHLEVEL and POPLEVEL commands.

```
PUSHLEVEL 'firstlevel'
PUSH PAGESIZE DECIMALS "Saves values in FIRSTLEVEL
       .
       .
       .
PUSHLEVEL 'secondlevel'
PUSH month product     "Saves values in SECONDLEVEL
       .
       .
       .
POPLEVEL 'secondlevel' "Restores values in SECONDLEVEL
       .
       .
       .
POPLEVEL 'firstlevel'  "Restores values in FIRSTLEVEL
```

Normally, you will not use more than one set of PUSHLEVEL and POPLEVEL commands in a single program. However, the nesting feature comes into play automatically when one program calls another program, and each program contains a set of PUSHLEVEL and POPLEVEL commands.

Using CONTEXT to Save Several Values at Once

As an alternative to using PUSHLEVEL and POPLEVEL, you can use the CONTEXT command. After you create a context, you can save the current status of dimensions and the values of options, single-cell variables, valuesets, and single-cell relations in the context. You can then restore some or all of the object values from the context. The CONTEXT function returns information about objects in a context.

Handling Errors

A well-designed program handles errors gracefully and reports each error in an informative way. The OLAP DML provides commands such as TRAP to help you detect and report errors in your programs.

How An Error Is Signaled

When an error occurs anywhere in a program, the error is signaled. To signal the error, the following actions are performed.

1. The name of the error is stored in the ERRORNAME option, and the text of the error message is stored in the ERRORTEXT option.
2. If ECHOPROMPT is YES, then the error message is sent to the current outfile or to the debugging file, when there is one.
3. If error trapping is off, then the execution of the program is halted. If error trapping is on, then the error is trapped.

How An Error Is Trapped

To make sure the program works correctly, you should anticipate errors and set up a system for handling them. You can use the TRAP command to turn on an error-trapping mechanism in a program. If error trapping is on when an error is signaled, then the execution of the program is not halted. Instead, error trapping does the following:

1. Turns off the error-trapping mechanism to prevent endless looping in case additional errors occur during the error-handling process
2. Branches to the label that is specified in the TRAP command
3. Executes the commands following the label

Handling Errors While Saving the Session Environment

To correctly handle errors that might occur while you are saving the session environment, place your PUSHLEVEL command before the TRAP command and your PUSH commands after the TRAP command.

```
PUSHLEVEL 'firstlevel'
TRAP ON error
PUSH . . .
```

In the abnormal exit section of your program, place the error label (followed by a colon) and the commands that restore the session environment and handle errors. The abnormal exit section might look like this.

```
error:
POPLEVEL 'firstlevel'
OUTFILE EOF
```

These commands restore saved dimension status and option values and reroute output to the default outfile.

Suppressing Error Messages

If you do not want to produce the error message that is normally provided for a given error, then you can use the `NOPRINT` keyword with the `TRAP` command.

```
TRAP ON error NOPRINT
```

When you use the `NOPRINT` keyword with `TRAP`, control branches to the `error` label, and an error message is not issued when an error occurs. The commands following the `error` label are then executed.

When you suppress the error message, you might want to produce your own message in the abnormal exit section. The `SHOW` command produces the text you specify but does not signal an error.

```
TRAP ON error NOPRINT
       .
       .
       .
error:
       .
       .
       .
SHOW 'The report will not be produced.'
```

The program continues with the next command after producing the message.

Identifying the Error That Occurred

All errors have names. Whenever an error is signaled, the error name is stored in the `ERRORNAME` option. If you want to perform one set of activities when one type of error occurs, and a different set of activities if another type of error occurs, then you can test the value of the `ERRORNAME` option. The `ERRORTEXT` option contains a description of the error.

Creating Your Own Error Messages

All errors that occur when commands or command sequences do not conform to its requirements are signaled automatically. In your program, you can establish additional requirements for your own application. When a requirement is not met, you can execute the SIGNAL command to signal an error.

You can give the error any name. When the SIGNAL command is executed, the error name you specify is stored in the ERRORNAME option, just as an error name is stored. If you specify your own error message in the SIGNAL command, then your message is produced just as an error message is produced. When you are using a TRAP command to trap errors, a SIGNAL command branches to the TRAP label after the error message is produced.

Example 7–14 Signaling an Error

Suppose your program produces a report that can present from one to nine months of data. You can signal an error when the program is called with an argument value greater than nine. In this example, nummonths is the name of the argument that must be no greater than nine.

```
select:
TRAP ON error
PUSH month
LIMIT month TO nummonths
IF STATLEN(month) GT 9
   THEN SIGNAL toomany -
     'You can specify no more than 9 months.'
REPORT DOWN district W 6 units
finish:
POP month
RETURN
error:
POP month
IF ERRORNAME EQ 'TOOMANY'
   THEN SHOW 'No report produced'
```

If you want to produce a warning message without branching to an error label, then you can use the SHOW command.

```
select:
LIMIT month TO nummonths
IF STATLEN(month) GT 9
   THEN DO
     SHOW 'You can select no more than 9 months.'
```

```
     GOTO finish
     DOEND
REPORT DOWN district W 6 units
finish:
POP month
RETURN
```

Handling Errors in Nested Programs

When you write a program that runs another program, the second program is nested within the first program. The second program might, in turn, run another nested program.

The error-handling section in each program should restore the environment. It can also handle any special error conditions that are particular to that program. For example, if your program signals its own error, then you can include commands that test for that error.

Any other errors that occur in a nested program should be passed up through the chain of programs and handled in each program. To pass errors through a chain of nested programs, you can use one of two methods, depending on when you want the error message to be produced:

- The error message is produced immediately, and the error condition is then passed through the chain of programs.
- The error is passed through the chain of programs first, and the error message is produced at the end of the chain.

The `SIGNAL` command is used in both methods.

Example 7–15 Producing the Error Message Immediately

To produce the error message immediately, use a `TRAP` command in each nested program, but do not use the `NOPRINT` keyword. When an error occurs, an error message is produced immediately, and execution branches to the trap label.

At the trap label, perform whatever error-handling commands you want and restore the environment. Then execute a `SIGNAL` command with the `PRGERR` keyword.

```
SIGNAL PRGERR
```

When you use the `PRGERR` keyword with the `SIGNAL` command, no error message is produced, and the name `PRGERR` is not stored in `ERRORNAME`. The `SIGNAL` command signals an error condition that is passed up to the program from which

the current program was run. If the calling program contains a trap label, then execution branches to that label.

When each program in a chain of nested programs uses the TRAP and SIGNAL commands in this way, you can pass the error condition up through the entire chain. Each program has commands like these.

```
TRAP ON error
   .
   .       "Body of program and normal exit commands
   .
RETURN
error:
   .
   .       "Error-handling and exit commands
   .
SIGNAL PRGERR
```

Example 7–16 Producing the Error Message at the End of the Chain

To produce the error message at the end of a chain of nested programs, use a TRAP command with the NOPRINT keyword. When an error occurs in a nested program, execution branches to the trap label, but the error message is suppressed.

At the trap label, perform whatever error-handling commands you want and restore the environment. Then execute the following SIGNAL command.

```
SIGNAL ERRORNAME ERRORTEXT
```

The ERRORNAME option contains the name of the original error, and the ERRORTEXT option contains the error message for the original error. The SIGNAL command shown above passes the original error name and error text to the calling program. If the calling program contains a trap label, then execution branches to that label.

When each program in a chain of nested programs uses the TRAP and SIGNAL commands in this way, the original error message is produced at the end of the chain. Each program has commands like these.

```
TRAP ON error NOPRINT
   .
   .       "Body of program and normal exit commands
   .
RETURN
```

```
error:
   .
   .       "Error-handling and exit commands
   .
SIGNAL ERRORNAME ERRORTEXT
```

Compiling Programs

You can explicitly compile a program by using the `COMPILE` command. If you do not explicitly compile a program, then it is compiled when you run the program for the first time.

When a program is compiled, it translates the program commands into efficient processed code that executes much more rapidly than the original text of the program. If errors are encountered in the program, then the compilation is not completed, and the program is considered to be uncompiled.

After you compile a program, the compiled code is used each time you run the program in the current session. When you update and commit your analytic workspace after compiling a program, the compiled code is saved in your analytic workspace and used to run the program in future sessions. Therefore, you should be sure to update and commit after compiling a program. This is particularly critical when the program is part of an application that is run by many users. Unless the compiled version of the program is saved in the analytic workspace, the program is recompiled individually in each user session.

Example 7–17 Using the COMPILE Command

The following is an example of a `COMPILE` command that compiles the `myprog` program.

```
COMPILE myprog
```

Suppose you misspell the dimension `month` in a `LIMIT` command in the `myprog` program.

```
LIMIT motnh TO LAST 6
```

When the `COMPILE` command encounters this command, it produces the following message.

```
ERROR: (MXMSERR00) Analytic workspace object MOTNH does not exist.
In DEMO!MYPROG PROGRAM:
limit motnh to last 6
```

You can edit the program to correct the error and then try to compile it again.

Finding Out If a Program Has Been Compiled

You can use the ISCOMPILED choice of the OBJ function to determine whether a specific program in your analytic workspace has been compiled since the last time it was modified. The function returns a Boolean value.

```
SHOW OBJ(ISCOMPILED 'myprogram')
```

Programming Methods That Prevent Compilation

Program lines that include ampersand substitution will not be compiled. Any syntax errors will not be caught until the program is run. A program whose other lines compiled correctly is considered to be a compiled program.

When your program defines an object and then uses the object in the program, the program will not compile. COMPILE treats the reference to the object as a misspelling because the object does not yet exist in the analytic workspace.

> **See Also:** "Passing Arguments as Text with Ampersand Substitution" on page 7-9 for information about ampersand substitution.

Testing and Debugging Programs

Even when your program compiles cleanly, you must also test the program by running it. Running a program helps you detect errors in commands with ampersand substitution, errors in logic, and errors in any nested programs.

To test a program by running it, use a full set of test data that is typical of the data that the program will process. To confirm that you test all the features of the program, including error-handling mechanisms, run the program several times, using different data and responses. Use test data that:

- Falls within the expected range
- Falls outside the expected range
- Causes each section of a program to execute

Generating Diagnostic Messages

Each time you run the program, confirm that the program executes its commands in the correct sequence and that the output is correct. As an aid in analyzing the execution of your program, you can include `SHOW` commands in the program to produce diagnostic or status messages. Then delete the `SHOW` commands after your tests are complete.

When you detect or suspect an error in your program or a nested program, you can track down the error by using the debugging techniques that are described in the the rest of this section.

Identifying Bad Lines of Code

When you set the `BADLINE` option to `YES`, additional information will be produced, along with any error message when a bad line of code is encountered. When the error occurs, the error message, the name of the program, and the program line that triggered the error are sent to the current outfile.

You can edit the specified program to correct the error and then run the original program.

Example 7–18 Using the BADLINE Option

In a simple program called `test`, the variable `myint1` is divided by zero.

```
DEFINE test PROGRAM
PROGRAM
VARIABLE myint1 INTEGER
VARIABLE myint2 INTEGER
myint1 = 0
myint2 = 250/myint1
END
```

If you run the program when the `DIVIDEBYZERO` option is set to `NO`, then an error occurs because division by zero is not allowed. When `BADLINE` is set to `YES`, the following messages are recorded in the current outfile.

```
ERROR: (MXXEQ01) A division by zero was attempted. Set DIVIDEBYZERO to
YES if you want NA to be returned as the result of division by zero.
In DEMO!TEST PROGRAM:
myint2 = 250/myint1
```

Sending Output to a Debugging File

If your program contains an error in logic, then the program might execute without producing an error message, but it will execute the wrong set of commands or produce incorrect results. For example, suppose you write a Boolean expression incorrectly in an IF command (for example, you use NE instead of EQ). The program will execute the commands you specified, but it will do so under the wrong conditions.

To find an error in program logic, you often need to see the order in which the commands are being executed. One way you can do this is to create a debugging file and then examine the file to diagnose any problems in your programs.

Creating a debugging file

To create a debugging file, you use the DBGOUTFILE command. The syntax of the DBGOUTFILE command is shown below.

```
DBGOUTFILE {EOF|[APPEND] file-id [NOCACHE]}
```

The command has the following arguments:

- The EOF keyword specifies that the current debugging file should be closed, and that debugging output should no longer be sent to a file.
- The APPEND keyword specifies that the output should be added to the end of an existing disk file. If you omit this argument and a file exists with the specified name, then the new output replaces the current contents of the file.
- The *file-id* argument specifies the name of the file to receive the debugging output.
- The NOCACHE keyword causes the OLAP DML to write to the debugging file each time it executes a line of code. Without this keyword, file I/O activity is reduced by saving text and writing it periodically to the file.

For more information about the DBGOUTFILE command, see the entry for the command in Oracle9i OLAP DML Reference help.

Specifying the contents of the debugging file

Using the DBGOUTFILE command merely creates a file for debugging. To specify that you want each program line to be sent, as it executes, to the debugging file, set the PRGTRACE option to YES.

If you want the debugging file to interweave the program lines with both the program's input and error messages, then set the ECHOPROMPT option to YES.

For the syntax of the ECHOPROMPT and PRGTRACE options, see the entry for each option in Oracle9i OLAP DML Reference help.

Example 7–19 Using a Debugging File

The following commands create a useful debugging file called debug.txt in the current directory alias.

```
prgtrace = yes
echoprompt = yes
dbgoutfile 'debug.txt'
```

After executing these commands, you can run your program as usual. To close the debugging file, execute this command.

```
dbgoutfile eof
```

In the following sample program, the first LIMIT command has a syntax error.

```
DEFINE ERROR_TRAP PROGRAM
PROGRAM
trap on traplabel
limit month to first badarg
limit product to first 3
limit district to first 3
report sales
traplabel:
signal errorname errortext
END
```

With PRGTRACE and ECHOPROMPT both set to YES and with DBGOUTFILE set to send debugging output to a file called debug.txt, the following text is sent to the debug.txt file when you execute the error_trap program.

```
(PRG= ERROR_TRAP)
(PRG= ERROR_TRAP) trap on traplabel
(PRG= ERROR_TRAP)
(PRG: ERROR_TRAP) limit month to first badarg
ERROR: BADARG does not exist in any attached database.
(PRG= ERROR_TRAP) traplabel:
(PRG= ERROR_TRAP) signal errorname errortext
ERROR: BADARG does not exist in any attached database.
```

8

Working with Models

This chapter describes how to use models to calculate data. It includes the following topics:

- Using Models to Calculate Data
- Creating a Nested Hierarchy of Models
- Basic Modeling Commands
- Compiling a Model
- Running a Model
- Debugging a Model
- Modeling for Multiple Scenarios

Using Models to Calculate Data

A **model** is a set of interrelated equations that can assign results either to a variable or to a dimension value. For example, in a financial model, you can assign values to specific line items, such as `gross.margin` or `net.income`.

```
gross.margin = revenue - cogs
```

If an = command assigns data to a dimension value or refers to a dimension value in its calculations, then it is called a dimension-based equation. A dimension-based equation does not refer to the dimension itself, but only to the values of the dimension. Therefore, if the model contains any dimension-based equations, then you must specify the name of each of these dimensions in a `DIMENSION` command at the beginning of the model.

If a model contains any dimension-based equations, then you must supply the name of a **solution variable** when you run the model. The solution variable is both a source of data and the assignment target of model equations. It holds the input data used in dimension-based equations, and the calculated results are stored in designated values of the solution variable. For example, when you run a financial model based on the `line` dimension, you might specify `actual` as the solution variable.

Dimension-based equations provide flexibility in financial modeling. Since you do not need to specify the modeling variable until you solve a model, you can run the same model with the `actual` variable, the `budget` variable, or any other variable that is dimensioned by `line`.

Example 8–1 Creating a Model

Suppose that you define a model, called `income.calc`, that will calculate line items in the income statement.

```
define income.calc model
ld Calculate line items in income statement
```

After defining the model, you can use the `MODEL` command or the OLAP Worksheet editor to specify the contents of the model. A model can contain `DIMENSION` commands, = commands, and comments. All the `DIMENSION` commands must

come before the first equation. For the current example, you can specify the lines shown in the following model.

```
DEFINE INCOME.CALC MODEL
LD Calculate line items in income statement
MODEL
DIMENSION line
net.income = opr.income - taxes
opr.income = gross.margin - (marketing + selling + r.d)
gross.margin = revenue - cogs
END
```

When you write the equations in a model, you can place them in any order. When you compile the model, either with the `COMPILE` command or by running the model, the order in which the model equations are solved is determined. If the calculated results of one equation are used as input to another equation, then the equations are solved in the order in which they are needed.

To run the `income.calc` model and use `actual` as the solution variable, you execute the following command.

```
income.calc actual
```

If the solution variable has dimensions other than the dimensions on which model equations are based, then a loop is performed automatically over the current status list of each of these "extra" dimensions. For example, `actual` is dimensioned by `month` and `division`, as well as by `line`. If `division` is limited to `ALL`, and `month` is limited to `OCT96` to `DEC96`, then the `income.calc` model is solved for the three months in the status for each of the divisions.

How Dimension Values Are Treated in a Model

If a model contains an = command that assigns data to a dimension value, then the dimension is limited temporarily to that value, performs the calculation, and then restores the initial status of the dimension.

For example, a model might have the following commands.

```
DIMENSION line
gross.margin = revenue - cogs
```

If you specify `actual` as the solution variable when you run the model, then the following code is constructed and executed.

```
PUSH line
LIMIT line TO gross.margin
actual = actual(line revenue) - actual(line cogs)
POP line
```

This behind-the-scenes construction lets you perform complex calculations with simple model equations. For example, line item data might be stored in the `actual` variable, which is dimensioned by `line`. However, detail line item data might be stored in a variable named `detail.data`, with a dimension named `detail.line`.

If your analytic workspace contains a relation between `line` and `detail.line`, which specifies the line item to which each detail item pertains, then you might write model equations such as the following ones.

```
revenue = total(detail.data line)
expenses = total(detail.data line)
```

The relation between `detail.line` and `line` is used automatically to aggregate the detail data into the appropriate line items. The code that is constructed when the model is run ensures that the appropriate total is assigned to each value of the `line` dimension. For example, while the equation for the `revenue` item is calculated, `line` is temporarily limited to `revenue`, and the `TOTAL` function returns the total of detail items for the `revenue` value of `line`.

Creating a Nested Hierarchy of Models

The `INCLUDE` command allows you to include one model within another model. A model can contain only one `INCLUDE` command. The `INCLUDE` command must come before any equations in the model, and it can specify the name of just one model to include. The model that contains the `INCLUDE` command is referred to as the **parent model**. The included model is referred to as the **base model**.

You can nest models by placing an `INCLUDE` command in a base model. For example, model `m1` can include model `m2`, and model `m2` can include model `m3`. The nested models form a hierarchy. In this example, `m1` is at the top of the hierarchy, and `m3` is at the root.

Working with the INCLUDE Command

If a model contains an INCLUDE command, then it cannot contain any DIMENSION commands. A parent model inherits its dimensions, if any, from the DIMENSION commands in the root model of the included hierarchy. In the example just given, models m1 and m2 both inherit their dimensions from the DIMENSION commands in model m3.

The INCLUDE command allows you to create modular models. If certain equations are common to several models, then you can place these equations in a separate model and include that model in other models as needed.

The INCLUDE command also facilitates what-if analyses. An experimental model can draw equations from a base model and selectively replace them with new equations. To support what-if analysis, you can use equations in a model to mask previous equations. The previous equations can come from the same model or from included models. A masked equation is not executed.

After you compile a model, either by running it or by using the COMPILE command, you can run an OLAP DML program called MODEL.COMPRPT to produce a report on the structure of the compiled model. If you run MODEL.COMPRPT after compiling a model that contains a masked equation, then you will find that the masked equation is not shown in the report.

Basic Modeling Commands

The following table lists the most common OLAP DML commands that you will use when you define and run models.

Command	Description
DEFINE	Adds a new model to an analytic workspace.
MODEL	Specifies completely new contents for a new or existing model.
DIMENSION	Lists one or more dimensions that are referred to in dimension-based equations in the model.
INCLUDE	Specifies a base model to include in the parent model.
=	Performs a calculation and assigns the result to a target. The target can be a variable or it can be represented by a dimension value.
COMPILE	Compiles a model without running it and saves the compiled code in the workspace. If you run a new or revised model without first compiling it, then the model is compiled automatically at that time.

Writing Equations in a Model

When you write the equations in a model, you should keep these points in mind:

- Within a single dimension-based equation, all the dimension values must belong to the same dimension.
- You cannot use ampersand substitution in model equations.

Writing DIMENSION and INCLUDE Commands

When you write `DIMENSION` and `INCLUDE` commands, you should keep these points in mind:

- Any `DIMENSION` commands or `INCLUDE` command must come before the first equation in a model.
- In the `DIMENSION` commands, you must list the names of all the dimensions on which model equations are based. In the following example, `gross.margin`, `revenue`, and `cogs` are values of the `line` dimension, so `line` is specified in a `DIMENSION` command.

  ```
  DIMENSION line
  gross.margin = revenue - cogs
  ```

- `DIMENSION` commands must also list any dimension that is an argument to a function that refers to a dimension value. In the following example, `month` must be specified in a `DIMENSION` command.

  ```
  DIMENSION line, month
  revenue = LAG(revenue, 1, month) * 1.05
  ```

- If a model contains an `INCLUDE` command, then it cannot contain any `DIMENSION` commands. The included model (or the root model in a hierarchy) must contain the `DIMENSION` commands needed by the parent model(s).
- If a model equation assigns results to a dimension value, then code is constructed that loops over the values of any of the other nontarget dimensions listed in the `DIMENSION` commands. The nontarget dimension listed first in the `DIMENSION` commands is treated as the slowest-varying dimension.
- A model will execute most efficiently when you observe the following guidelines for coordinating the dimensions in `DIMENSION` commands and the dimensions of the solution variable:

- List the target dimension of the model as the *first* dimension in the DIMENSION commands and as the *last* dimension in the definition of the solution variable.
- In DIMENSION commands, list the nontarget dimensions in the *reverse* order of their appearance in the definition of the solution variable. This means that the nontarget dimensions will have the same order in the model and in the solution variable in terms of fastest-varying and slowest-varying dimension.

- If the solution variable has dimensions that are not used or referred to in model equations, then do not include them in DIMENSION commands.
- If your analytic workspace contains a variable whose name is the same as a dimension value, or if the same dimension value exists in two different dimensions, then there could be ambiguities in your model equations. Since you can use a variable and a dimension value in exactly the same way in a model equation, a name might be the name of a variable, or it might be a value of any dimension in your analytic workspace.
- Your DIMENSION commands are used to determine whether each name reference in an assignment statement (that is, the = command) is a variable or a dimension value. "Compiling a Model" on page 8-7 explains how the name references are resolved.

Compiling a Model

When you finish writing the commands in a model, you can use the COMPILE command to compile the model. During compilation, COMPILE checks for format errors, so you can use COMPILE to help debug your code before running a model. If you do not use the COMPILE command before you run the model, then the model will be compiled automatically before it is solved.

When you compile a model, either by using the COMPILE command or by running the model, the model compiler examines each equation to determine whether the assignment target and each data source is a variable or a dimension value.

To resolve each name reference, the following procedure is used.

1. The dimensions in the DIMENSION commands are searched, in the order they are listed, to determine whether the name matches a dimension value of a listed dimension. The search concludes as soon as a match is found.
2. If the name does not match a value of a listed dimension, then the variables in the attached analytic workspaces are searched to find a match.

After resolving each name reference, the model compiler analyzes dependencies between the equations in the model. A dependence exists when the expression on the right-hand side of the equal sign in one equation refers to the assignment target of another equation. If an = command indirectly depends on itself as the result of the dependencies among equations, then a cyclic dependence exists between the equations.

The model compiler structures the equations into blocks and orders the equations within each block, and the blocks themselves, to reflect dependencies. The compiler can produce three types of solution blocks: simple blocks, step blocks, and simultaneous blocks.

Simple Blocks

Simple blocks include equations that are independent of each other and equations that have dependencies on each other that are noncyclic.

If a block contains equations that solve for values A, B, and C, then a noncyclic dependence can be illustrated as shown below where the arrows indicate that A depends on B, and B depends on C.

Step Blocks

Step blocks include equations that have a cyclic dependence that is a one-way dimensional dependence. A **dimensional dependence** occurs when the data for the current dimension value depends on data from previous or later dimension values. The dimensional dependence is one way when the data depends on previous values only or later values only, but not both.

Dimensional dependence typically occurs over a time dimension. For example, it is common for a line item value to depend on the value of the same line item or a different line item in a previous time period. If a block contains equations that solve for values A and B, then a one-way dimensional dependence can be illustrated as shown in the figure below where arrows indicate that A depends on B, and B depends on the value of A from a previous time period.

Simultaneous Blocks

Simultaneous blocks include equations that have a cyclic dependence that is other than one-way dimensional. The cyclic dependence may be two-way dimensional, or it may involve no dimensional qualifiers at all.

An example of a cyclic dependence that is two-way dimensional can be illustrated as shown below where the arrows indicate that A depends on the value of B from a future period, while B depends on the value of A from a previous period.

A ⟶ LEAD(B) ⟶ LAG(A)

An example of a cyclic dependence that does not depend on any dimensional qualifiers can be illustrated as shown below where the arrows indicate that A depends on B and B depends on A.

A ⟶ B ⟶ A

Running a Model

When you run a model, you should keep these points in mind:

- Before you run a model, the input data must be available in the solution variable. For example, before running the `income.calc` model (shown earlier in this chapter) with `actual` as the solution variable, you must have current data in the `revenue`, `cogs`, `marketing`, `selling`, `r.d`, and `taxes` line items of `actual`.
- Before running a model that contains a block of simultaneous equations, you might want to check or modify the values of some OLAP DML options that control the solution of simultaneous blocks. Simultaneous equations are discussed in the section entitled "Solving Simultaneous Equations" on page 8-10.
- If your model contains any dimension-based equations, then you must provide a numeric solution variable that serves both as a source of data and as the assignment target for equation results. The solution variable is usually dimensioned by all the dimensions on which model equations are based, and it can have “extra” dimensions as well.
- When you run a model, a loop is performed automatically over the values in the current status list of each of the extra dimensions of the solution variable.

- If a model equation bases its calculations on data from previous time periods (for example, if you use a LAG function), then the solution variable must contain data for these previous periods. If it does not, or if the first value of the time dimension is in the status, then the results of the calculation will be NA.

Using Data from Past and Future Time Periods

Several OLAP DML functions make it easy for you to use data from past or future time periods. For example, the LAG function returns data from a specified previous time period, and the LEAD function returns data from a specified future period. The Oracle9i OLAP DML Reference help lists some built-in functions that are useful in analyzing financial data.

When you run a model that uses past or future data in its calculations, you must make sure that your solution variable contains the necessary past or future data. For example, a model might contain an assignment statement (that is, the = command) that bases an estimate of the revenue line item for the current month on the revenue line item for the previous month.

```
DIMENSION line month
.
.
.
revenue = LAG(revenue, 1, month) * 1.05
```

If the month dimension is limited to apr96 to jun96 when you run the model, then you must be sure that the solution variable contains revenue data for mar96.

If your model contains a LEAD function, then your solution variable must contain the necessary future data. For example, if you want to calculate data for the months of April through June of 1996, and if the model retrieves data from one month in the future, then the solution variable must contain data for July 1996 when you run the model.

Solving Simultaneous Equations

An iterative method is used to solve the equations in a simultaneous block. In each iteration, a value is calculated for each equation, and compares the new value to the value from the previous iteration. If the comparison falls within a specified tolerance, then the equation is considered to have converged to a solution. If the comparison exceeds a specified limit, then the equation is considered to have diverged.

If all the equations in the block converge, then the block is considered solved. If any equation diverges or fails to converge within a specified number of iterations, then the solution of the block (and the model) fails and an error occurs.

You can use OLAP DML options to exercise control over the solution of simultaneous equations. For example, you can specify the solution method to use, the factors to use in testing for convergence and divergence, the maximum number of iterations to perform, and the action to take when the = command diverges or fails to converge.

Debugging a Model

The OLAP DML provides an assortment of tools that will help you debug your models. These tools are listed in the following table.

Tool	Purpose
MODTRACE	An option that controls whether each line of a model is sent to the current outfile while you run the model. When MODTRACE is set to YES, the model lines are shown, and you can observe the order in which the equations are solved.
DBGOUTFILE	A command that creates a debugging file to which the MODTRACE output is sent. For information about this command see "Sending Output to a Debugging File" on page 7-31.
MODEL.COMPRPT	A program that produces a report on the structure of a compiled model. The report shows how model equations are grouped into blocks.
MODEL.DEPRPT	A program that produces a report on the dependencies in model equations. The report lists the assignment target and data sources for each equation and specifies any dimensions of the dependencies in the equation.
MODEL.XEQRPT	A program that produces a report on the solution status of a model. If the model contains simultaneous equations, then the report specifies the values of the options that control simultaneous solutions.
INFO	A function that lets you obtain specific information about a model that you have compiled or executed.

Modeling for Multiple Scenarios

Instead of calculating a single set of figures for a month and division, you might want to calculate several sets of figures, each based on different assumptions.

You can define a **scenario** model that calculates and stores forecast or budget figures based on different sets of input figures. For example, you might want to calculate profit based on “optimistic,” “pessimistic,” and “best-guess” figures.

Building a Scenario Model

To build a scenario model, follow these steps.

1. Define a scenario dimension.
2. Define a solution variable dimensioned by the scenario dimension.
3. Enter input data into the solution variable.
4. Write a model to calculate results based on the input data.

Suppose, for example, you want to calculate profit figures based on optimistic, pessimistic, and best-guess revenue figures for each division. The steps for building this scenario model are explained in the following example.

Example 8–2 Building a Scenario Model

You can call the scenario dimension `scenario` and give it values that represent the scenarios you want to calculate.

These commands give `scenario` the values `optimistic`, `pessimistic` and `bestguess`.

```
DEFINE scenario DIMENSION TEXT
LD Names of scenarios
MAINTAIN scenario ADD optimistic pessimistic bestguess
```

These commands create a variable named `plan` dimensioned by three other dimensions (`month`, `line`, and `division`) in addition to the `scenario` dimension.

```
DEFINE plan DECIMAL <month line division scenario>
LD Scenarios for financials
```

For this example, you need to enter input data, such as revenue and cost of goods sold, into the `plan` variable.

For the best-guess data, you can use the data in the `budget` variable. Limit the `line` dimension to the input line items, and then copy the `budget` data into the `plan` variable.

```
LIMIT scenario TO 'BESTGUESS'
LIMIT line TO 'REVENUE' 'COGS' 'MARKETING' 'SELLING' 'R.D'
plan = budget
```

You might want to base the optimistic and pessimistic data on the best-guess data. For example, optimistic data might be 15 percent higher than best-guess data, and pessimistic data might be 12 percent less than best-guess data. With `line` still limited to the input line items, execute the following commands.

```
plan(scenario 'OPTIMISTIC') = 1.15 * plan(scenario 'BESTGUESS')
plan(scenario 'PESSIMISTIC') = .88 * plan(scenario 'BESTGUESS')
```

The final step in building a scenario model is to write a model that calculates results based on input data. The model might contain calculations very similar to those in the `budget.calc` model shown earlier in this chapter.

You can use the same equations for each scenario or you can use different equations. For example, you might want to calculate the cost of goods sold and use a different constant factor in the calculation for each scenario. To use a different constant factor for each scenario, you can define a variable dimensioned by `scenario` and place the appropriate values in the variable. If the name of your variable is `cogsval`, then your model might include the following equation for calculating the `cogs` line item.

```
cogs = cogsval * revenue
```

By using variables dimensioned by `scenario`, you can introduce a great deal of flexibility into your scenario model.

Similarly, you might want to use a different constant factor for each division. You can define a variable dimensioned by `division` to hold the values for each division. For example, if labor costs vary from division to division, then you might dimension `cogsval` by `division` as well as by `scenario`.

When you run your model, you specify `plan` as the solution variable. For example, if your model is called `scenario.calc`, then you solve the model with this command.

```
scenario.calc plan
```

A loop is performed automatically over the current status list of each of the dimensions of `plan`. Therefore, if the `scenario` dimension is limited to `ALL` when you run the `scenario.calc` model, then the model is solved for all three scenarios: `optimistic`, `pessimistic`, and `bestguess`.

9

Allocating Data

This chapter describes how to use the `ALLOCATE` command to allocate data from a source to a target variable. This chapter includes the following topics:

- Introduction to Allocation
- Preparing for an Allocation
- Creating an Aggregation Map for Allocation
- Using the Allocation Operators and Arguments

Introduction to Allocation

The Oracle OLAP `ALLOCATE` command distributes data from a source object to the cells of a target. The target is a variable that is dimensioned by one or more hierarchical dimensions. The source data is specified by dimension values at a higher level in a hierarchical dimension than the values that specify the target cells.

`ALLOCATE` uses an aggregation map to specify the dimensions and the values of the hierarchies to use in the allocation, the method of operation to use for a dimension, and other aspects of the allocation.

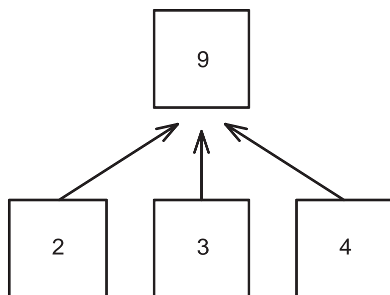
Some of the allocation operations are based on existing data. The object containing that data is the basis object for the allocation. In those operations, `ALLOCATE` distributes the data from the source based on the values of the basis object.

Forecasting and budgeting systems often use allocation in operations such as the automatic distribution of a bonus pool with the amounts based on the current salary and performance ratings of the employee.

An allocation is the opposite of an aggregation that you perform with the `AGGREGATE` command. In an aggregation, the data at lower levels of a hierarchy is combined into data at higher levels. In an allocation, data at a higher level in the hierarchy is distributed to lower levels.

The `ALLOCATE` command has operations that are the inverse of the operations of the `AGGREGATE` command. Figure 9–1 shows an aggregation up a simple hierarchy. In a `SUM` operation, the aggregation adds the detail level values 2, 3, and 4 to derive the value 9 at the aggregate level.

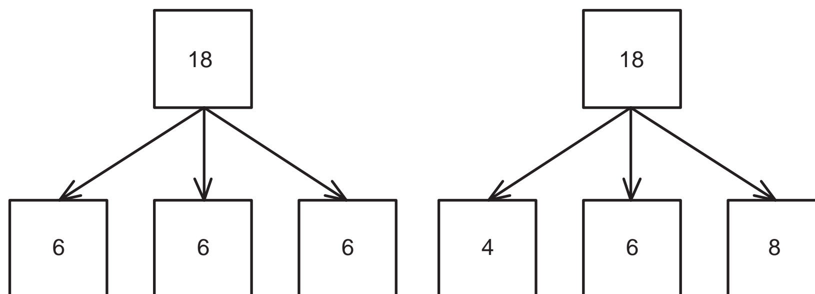
Figure 9–1 Aggregation in a Simple Hierarchy



As an example of an allocation, you could take the aggregate value 9, double it to 18, and allocate the results to the detail level with the previous values of the detail level cells as the basis of the allocation. In Figure 9–2 the hierarchy on the left shows

the result of an EVEN allocation operation, in which the source value is distributed evenly. Each detail level cell receives a value of 6. The hierarchy on the right shows a PROPORTIONAL allocation operation, in which the source value is distributed proportionately. The values allocated to the detail level are 4, 6 and 8.

Figure 9–2 Allocation in a Simple Hierarchy



The allocation operation methods range from simple allocations, such as copying the source data to the cells of the target variable, to very complex allocations, such as a proportional distribution of data from a source that is a formula, with the amount distributed being based on another formula, with multiple variables as targets, and with an aggregation map that specifies different methods of allocation for different dimensions.

The Oracle OLAP allocation system is very flexible and has many features, including the following:

- The source, basis, and target objects can be the same variable or they can be different objects.
- The source and basis objects can be formulas, so you can perform computations on existing data and use the result as the source or basis of the allocation.
- You can specify the method of operation of the allocation for a dimension. The operations range from simple to very complex. The operations of the allocation system are the inverse of the aggregation operations.
- You can specify whether the allocated value is added to or replaces the existing value of the target cell.
- You can specify an amount to add to or multiply by the allocated value before the result is assigned to the target cell.

- You can lock individual values in a dimension hierarchy so that the data of the target cells for those dimension values is not changed by the allocation. If you lock a dimension value, then the allocation system normalizes the source data, which subtracts the locked data from the source before the allocation. You can choose to not normalize the source data.
- You can specify minimum, maximum, floor, or ceiling values for certain operations.
- You can copy the allocated data to a second variable so that you can have a record of individual allocations to a cell that is the target of multiple allocations.
- You can specify ways of handling allocations when the basis has a null value.
- You can use the same aggregation map in different ALLOCATE commands to use the same set of dimension hierarchy values, operations, and arguments with different source, basis, or target objects.

Along with the ALLOCATE command, the OLAP DML contains other commands that support allocation. Table 9–1 lists those commands.

Table 9–1

Command	**Description**
AGGMAPINFO command	Returns information about the contents of an aggregation map object, such as the type of the aggregation map, which indicates whether it contains commands for aggregation or allocation.
ALLOCATE command	Allocates data from a source object to a target variable.
ALLOCERRLOGFORMAT command	Determines the contents and the formatting for the error log that you specify with the ERRORLOG argument to the ALLOCATE command.
ALLOCERRLOGHEADER command	Determines the column headings for the error log.
ALLOCMAP command	Adds contents to an aggmap object that specify the path of the allocation down a dimension hierarchy, the method of operation, and other aspects of the allocation. Marks the object as an ALLOCMAP type aggregation map.
POUTFILEUNIT option	Identifies a location that receives information on the progress of an ALLOCATE command.

The remainder of this chapter describes the objects that you use to allocate data and the types of allocation operations, and provides examples of various allocations.

Preparing for an Allocation

To prepare for allocating data, you decide on the data to allocate, the data on which to base the allocation, and the variable cells to which you want to assign the allocated data. You determine or create objects that contain or produce the data.

The target must be a variable that is dimensioned by one or more hierarchical dimensions. The source is a variable or a formula and the basis is a formula, a relation, or a variable. The source, basis, and target can all be the same variable.

You also decide whether to use a target log object to keep a copy of the allocation and whether to specify a file to log errors. For a target log, you use a variable and for the error log you specify a fileunit.

You create an aggregation map to use for the allocation. The contents of the aggregation map specify the dimensions and the values of the dimension hierarchies to use in the allocation, the method of operation, and other aspects of the allocation.

You specify the values of the dimension hierarchies to use in the allocation with a relation object. If you do not already have a relation that specifies the child-parent relations from the hierarchy that you want to use in the allocation, then you create a self-relation, which is a relation on the dimension dimensioned by the same dimension. The child-parent relationships that you assign to the relation specify the path of the allocation.

Creating an Aggregation Map for Allocation

An aggregation map for allocation contains commands that specify relations that define the path of an allocation through a dimension hierarchy, the method of the allocation operation, and other aspects of the allocation. To create an aggregation map you define an aggmap object with the `DEFINE` command or specify an existing aggregation map with the `CONSIDER` command. You then use the `ALLOCMAP` command to add commands to the aggregation map and to mark it as an `ALLOCMAP` type aggregation map.

You can add the following commands to an `ALLOCMAP` aggregation map:

- One or more `RELATION` commands
- A `CHILDLOCK` command
- A `DEADLOCK` command
- A `DIMENSION` command

- An ERRORLOG command
- An ERRORMASK command
- A SOURCEVAL command

A RELATION command specifies a self-relation that identifies the child-parent relationships of a dimension hierarchy that you want to use in the allocation. You use a separate RELATION command for each dimension that you want to participate in the allocation. The order of the RELATION commands in the aggregation map determines the order of the allocation.

With a RELATION command you specify an operator that determines the method of the allocation for the hierarchy. The RELATION command also has arguments with which you can specify other aspects of an allocation. For example, you can use the ARGS MIN *minval* argument to specify a value that the allocation assigns to the target cell if the allocated value is below a minimum value. You can use the ARGS ADD argument to specify that the allocation adds the allocated data to the current data of the target cell before assigning the result to the cell instead of replacing the current data with the allocated data.

The CHILDLOCK command tells the ALLOCATE command whether to determine if RELATION commands in the aggregation map specify locks on both a parent and a child element of a dimension hierarchy.

The DEADLOCK command tells the ALLOCATE command whether to continue an allocation if it encounters a deadlock, which occurs when the allocation cannot distribute a value because the targeted cell is locked or, for some operations, has a basis value of NA.

The DIMENSION command specifies a single value to set as the status of a dimension that is not shared by the target variable and the source or the basis objects.

The ERRORLOG command specifies how many errors to allow in the error log specified by the ALLOCATE command and whether to continue the allocation if the maximum number of errors has occurred.

The ERRORMASK command specifies which error conditions to exclude from the error log.

The SOURCEVAL command specifies whether ALLOCATE changes the source data value after the allocation. You use SOURCEVAL only if the source is a variable. With SOURCEVAL you can specify that the value of the source after the allocation is zero or NA or the current value, which is the value that the cell had before the allocation.

In a recursive allocation, ALLOCATE applies the value specified by SOURCEVAL to any cell used as a source in the allocation.

Using the Allocation Operators and Arguments

With the OPERATOR argument to a RELATION command in an aggregation map for allocation, you must specify a method of operation for the allocation. The methods of operation fall into the following categories:

- Copy operators
- Even distribution operators
- Proportional distribution operator

The copy operators are COPY, HCOPY, MIN, MAX, FIRST, LAST, HFIRST, and HLAST. The even distribution operators are EVEN and HEVEN, and the proportional distribution operator is PROPORTIONAL.

The H versions of the operators are hierarchical operators that allocate data based on the hierarchical relationships specified in the relation object. The nonhierarchical operators, such as COPY and EVEN, do not assign a value to a target cell if the basis value for that cell is NA.

The hierarchical operators do not use basis values. Instead, they allocate data to all of the values in the dimension hierarchy specified by the relation even if the existing value of the target cell is NA. You must use the hierarchical operators carefully because they assign values to cells that have an NA basis and can therefore cause a huge increase in the detail level data.

With a RELATION command, you can also use the ARGS keyword to specify arguments that affect the allocation. With the arguments you can specify the following:

- A minimum or maximum value for the allocation to assign to the target cells.
- A floor or ceiling value so that if the allocated value is below the floor or above the ceiling value, then ALLOCATE assigns NA to the target cell.
- Whether to add the allocated value to the existing value or to replace the existing value with the allocated value.
- Locks on the cells of the target variable that are specified by the dimension values in a valueset object. The PROTECT argument protects the existing values of the cells and prevents them from being targets of the allocation. You can also specify whether the locked cell can be a source in the allocation. For example, if

the valueset specifies a dimension value that is at an intermediate level in the dimension hierarchy and you use the `WRITE` keyword, then `ALLOCATE` uses the locked value as the source that it allocates down the hierarchy. If you use the `READWRITE` keyword, then `ALLOCATE` does not continue the allocation down that branch of the hierarchy. You can also specify whether to normalize the source value, which subtracts the locked value from the source before the allocation.

- A weighting object that specifies a value that `ALLOCATE` adds to or multiplies by the allocated value before assigning the resulting value to the target cell. You can also specify whether to fill an NA value before applying the weighting factor.

Using the HEVEN and MAX Operators and the ADD Argument

The `HEVEN` operator allocates source data evenly to the target cells without considering a basis value. The `MAX` operator allocates the source value to the target cell that corresponds to the highest basis value. Example 9–1 demonstrates the use of these operators and of the `ADD` argument in a multidimensional allocation. The allocation path is directly from higher to lower values in the dimension hierarchies, with no allocation to intermediate hierarchy values.

The `fcstunits` variable is dimensioned by the hierarchical dimensions `time`, `geog`, and `product`. The dimensions are limited to one product, a few cities and regions, and the year 2002 and four months of 2002.

The cells of `fcstunits` that are dimensioned by the lower hierarchical dimension values, which are the cities and the months, have values assigned to them. Those values are forecasts of the number of product units to ship to those cities in those months. In the cells dimensioned by the higher hierarchical dimension values, which are the `YEAR02` and the region values, are additional product units to allocate to the cities and months.

A report of the `fcstunits` variable produces the following.

```
PRODUCT: SHORTS - BOYS
               -----------------FCSTUNITS-----------------
               -------------------TIME--------------------
GEOG             YEAR02     JUN02      JUL02      AUG02
-------------- ---------- ---------- ---------- ----------
EAST                  755         NA         NA         NA
WEST                  515         NA         NA         NA
CENTRAL               625         NA         NA         NA
BOSTON                 NA      5,760      5,690      4,750
ATLANTA                NA      7,600      8,520      7,300
```

```
CHICAGO                NA      4,660      4,840      5,120
DALLAS                 NA      8,380      9,380      8,150
DENVER                 NA      5,400      6,080      5,170
SEATTLE                NA      7,210      7,490      7,310
```

The `geogcityreg` relation relates the city values to the regions. The `timemonthyear` relation relates the month values to the year. Reports of the relations produce the following.

```
GEOG      GEOGCITYREG
--------- -----------
EAST      NA
WEST      NA
CENTRAL   NA
BOSTON    EAST
ATLANTA   EAST
CHICAGO   CENTRAL
DALLAS    CENTRAL
DENVER    WEST
SEATTLE   WEST

TIME    TIMEMONTHYEAR
------  -------------
YEAR02  NA
JUN02   YEAR02
JUL02   YEAR02
AUG02   YEAR02
```

The first `RELATION` command in the `xunitsalloc` aggregation map specifies that the first allocation occurs down the `geog` dimension hierarchy specified by the `geogcityreg` relation. The allocation evenly divides the values from the cells dimensioned by `YEAR02` and the region values and assigns the results to the children of the regions. The `REMOPERATOR LAST` keywords assign any remainder from the division to the last cell.

The values allocated to the regions in the first allocation do not appear in the report of the variable after the `ALLOCATE` command completes because the `SOURCEVAL` command in `xunitsalloc` specifies that `ALLOCATE` assign a zero value to cells that contained source values for the allocation. The region data allocated to the cities for `YEAR02` is `BOSTON 377`, `ATLANTA 378`, `CHICAGO 312`, `DALLAS 313`, `DENVER 257`, and `SEATTLE 258`.

The second `RELATION` command in `xunitsalloc` specifies that a second allocation occur down the `time` dimension hierarchy specified by the `timemonthyear` relation. The source values of the allocation are the values of the

cells dimensioned by YEAR02 and a city value. The allocation assigns each source value to the month that has the highest value for that city.

The ALLOCATE command in the example specifies only the fcstunits variable. Therefore, that variable is the source, the basis, and the target of the allocation. The command also specifies that the allocation use the xunitsalloc aggregation map.

Example 9–1 A Multidimensional Allocation Using the HEVEN and MAX Operators

```
LIMIT product TO 'SHORTS - BOYS'
LIMIT geog TO 'EAST' 'WEST' 'CENTRAL' -
  'BOSTON' 'ATLANTA' 'CHICAGO' 'DALLAS' 'DENVER' 'SEATTLE'
LIMIT time TO 'YEAR02' 'JUN02' TO 'AUG02'
DEFINE xunitsalloc AGGMAP
ALLOCMAP JOINLINES( -
  'RELATION geogstcity OPERATOR HEVEN REMOPERATOR LAST' -
  'RELATION timemonthyear OPERATOR MAX ARGS ADD' -
  'SOURCEVAL ZERO')
ALLOCATE fcstunits USING xunitsalloc
REPORT fcstunits
```

The REPORT of the fcstunits variable after the allocation produces the following.

```
PRODUCT: SHORTS - BOYS
               -----------------FCSTUNITS-----------------
               -------------------TIME--------------------
GEOG             YEAR02     JUN02      JUL02      AUG02
-------------- ---------- ---------- ---------- ----------
EAST                    0         NA         NA         NA
WEST                    0         NA         NA         NA
CENTRAL                 0         NA         NA         NA
BOSTON                  0      6,137      5,690      4,750
ATLANTA                 0      7,600      8,898      7,300
CHICAGO                 0      4,660      4,840      5,432
DALLAS                  0      8,380      9,693      8,150
DENVER                  0      5,400      6,337      5,170
SEATTLE                 0      7,210      7,748      7,310
```

Using the COPY Operator and the PROTECT Argument

Example 9–2 demonstrates the recursive copying of source data that is specified by a parent in a dimension hierarchy. The data is allocated to children of the parent and then that allocated data is the source of the allocation to the children of those children. It also demonstrates a second allocation in which different source data is copied to only one child and to its children.

The unitcost variable is dimensioned by time and prodid. The prodid dimension is a NUMBER dimension that has product identification numbers as values. The first LIMIT command sets the status of the prodid dimension to one value. The next LIMIT command sets the status of the time dimension to the year 2002, the first two quarters of 2002, and the first six months of 2002.

The YEAR02 cell of unitcost for the product is assigned the source value. A report of unitcost produces the following.

```
               -UNITCOST-
               --PRODID--
TIME             45285
-------------- ----------
YEAR02              34.25
Q1.02                  NA
Q2.02                  NA
JAN02                  NA
FEB02                  NA
MAR02                  NA
APR02                  NA
MAY02                  NA
JUN02                  NA
```

Example 9–2 defines the costalloc aggregation map and adds contents to it with the ALLOCMAP command. The RELATION command specifies the timeparent relation as the path for the allocation and the HCOPY operator as the method. The timeparent relation relates the children in the time dimension hierarchy to their parents.

The ALLOCATE command uses the unitcost variable as the source and the target of the allocation. Because the method is HCOPY, the allocation does not use a basis object.

A report of unitcost after the first allocation produces the following.

```
               -UNITCOST-
               --PRODID--
TIME             45285
-------------- ----------
YEAR02              34.25
Q1.02               34.25
Q2.02               34.25
JAN02               34.25
FEB02               34.25
MAR02               34.25
APR02               34.25
```

```
MAY02               34.25
JUN02               34.25
```

The example then changes the source value for `YEAR02`. It defines a valueset and limits the value of it to `Q1.02`.

The second `ALLOCMAP` command changes the contents of the aggregation map. The `RELATION` command specifies the same relation and `COPY` operation but it also specifies the `PROTECT` argument. The `SOURCEVAL` command specifies that the cells that contained source data are assigned a value of zero after the data is allocated.

The second allocation copies the value from the `YEAR02` cell, but it locks the `Q1.02` child and its children so that only the `Q2.02` child and its children receive the allocated value.

A report of `unitcost` after the second allocation produces the following.

```
               -UNITCOST-
               --PRODID--
TIME             45285
-------------- ----------
YEAR02               0.00
Q1.02               34.25
Q2.02               35.00
JAN02               34.25
FEB02               34.25
MAR02               34.25
APR02               35.00
MAY02               35.00
JUN02               35.00
```

Example 9–2 Using the COPY Operator with the PROTECT Argument

```
LIMIT prodid TO 45285
LIMIT time TO 'YEAR02' 'Q1.02' 'Q2.02' 'JAN02' TO 'JUN02'

unitcost(time 'YEAR02' prodid 45285) = 34.25

DEFINE costalloc AGGMAP
ALLOCMAP 'RELATION timeparent OPERATOR HCOPY'
ALLOCATE unitcost USING costalloc

unitcost(time 'YEAR02' prodid 45285) = 35.00

DEFINE lvset VALUESET time
LIMIT lvset TO 'Q1.02'
```

```
CONSIDER costalloc
ALLOCMAP JOINLINES( -
  'RELATION timeparent OPERATOR COPY ARGS PROTECT NONORMALIZE lvset' -
  'SOURCEVAL ZERO')
ALLOCATE unitcost USING costalloc
```

Using the HFIRST and HLAST Operators

Use the HFIRST and HLAST operators when you want to allocate data to the first or last child of a parent without considering a basis value. Example 9–3 assigns cash balance forward and cash forward data from the actual variable to the budget variable and then allocates the data from the budget cell specified by a parent time value to one specified by a child time value.

The actual and budget variables are dimensioned by the time, line, and product dimensions. The timeparent relation relates values of children in the time dimension to their parents.

The first LIMIT commands set the status of the time and line dimensions and limit the product dimension to one value. A report of the actual variable with that dimension status produces the following.

```
PRODUCT: DRESSES - WOMEN
          -----------------------ACTUAL-------------------------
          -------------------------TIME-------------------------
LINE        Q4.01      JAN02      FEB02      MAR02      Q1.02
--------- ---------- ---------- ---------- ---------- ----------
CASH B/F    1,000.00         NA         NA         NA         NA
CASH MVT      500.00         NA         NA         NA         NA
CASH C/F    1,500.00         NA         NA         NA         NA
```

The source data for the allocation is assigned from the cash forward line of the actual variable to the cash balance forward line of the budget variable. The next LIMIT command limits the line dimension to CASH B/F to restrict the allocation to that value. Example 9–3 then defines an aggregation map and adds contents to it with the ALLOCMAP command. The contents are a single RELATION command that specifies the HFIRST operator. The ALLOCATE command allocates the data from the Q1.02 parent to its first child, JAN02.

Forecasting a fifty per cent increase in the cash forward amount by the end of the quarter, the example multiplies by 1.5 the value from the Q4.01 cash forward line of the actual variable and assigns the result to the Q1.02 cash forward line of the

`budget` variable. The `CONSIDER` and `ALLOCMAP` commands change the contents of the aggregation map so that the `RELATION` command specifies the `HLAST` operator.

The `line` dimension is limited to cash forward and then the `ALLOCATE` command allocates the data from the `Q1.02` parent to its last child, `MAR02`. Finally, the last `LIMIT` command resets the status of the `line` dimension. A report of the `budget` variable after the allocation produces the following.

```
PRODUCT: DRESSES - WOMEN
          ------------------------BUDGET------------------------
          -------------------------TIME-------------------------
LINE        Q4.01      JAN02      FEB02      MAR02      Q1.02
--------- ---------- ---------- ---------- ---------- ----------
CASH B/F          NA   1,500.00         NA         NA   1,500.00
CASH MVT          NA         NA         NA         NA         NA
CASH C/F          NA         NA         NA   2,250.00   2,250.00
```

Example 9–3 Allocating Data to the First and Last Children of a Parent

```
LIMIT time TO 'Q4.01' 'JAN02' TO 'MAR02' 'Q1.02'
LIMIT line TO 'CASH B/F' 'CASH MVT' 'CASH C/F'
LIMIT product TO 'DRESSES - WOMEN'

" Assign the value of actual Q4.01 CASH C/F to budget Q1.02 CASH B/F
budget(time 'Q1.02' line 'CASH B/F') = actual(time 'Q4.01' line 'CASH C/F')

LIMIT line TO 'CASH B/F'

DEFINE qtomalloc AGGMAP
ALLOCMAP 'RELATION timeparent OPERATOR HFIRST'

" Allocate the Q1.02 value to the first month of the quarter
ALLOCATE budget USING qtomalloc

" Forecast a 50% increase in cash forward by the end of the quarter
budget(time 'Q1.02' line 'CASH C/F') = actual(time 'Q4.01' line 'CASH C/F') * 1.5

CONSIDER qtomalloc
ALLOCMAP 'RELATION timeparent OPERATOR HLAST'

LIMIT line TO 'CASH C/F'

" Allocate the Q1.02 value to the last month of the quarter
ALLOCATE budget USING qtomalloc

LIMIT line TO 'CASH B/F' 'CASH MVT' 'CASH C/F'
```

Using the PROPORTIONAL Operator

The PROPORTIONAL operator allocates source data proportionately to the target cells based on the values of the basis object. Example 9–4 demonstrates two proportional allocations of data recursively down the time dimension hierarchy.

The actual and budget variables are dimensioned by the time, line, and product dimensions. The timeparent relation relates values of children in the time dimension to their parents.

The first allocation allocates a forecasted revenue value from YEAR02 to the quarters and then to the months of that year. The allocation is based on the revenue from the same time periods for the previous year. Actual values for the first quarter of 2002 are then assigned to the cells of the budget variables. The second allocation locks the budget cells for the first quarter and its children, normalizes the source value by subtracting the locked quarter value from the source, and then allocates the remaining value to the other quarters and their children.

The first LIMIT commands set the status of each of the line and product dimensions to one value and limit the product dimension to the year, quarter, and month values for 2002.

The budget variable for 2002 has values that were copied from the actual variable for 2001. The example does not include that operation. The forecasted total revenue value for the product for the year 2002 is assigned to the budget variable. That value is calculated to be ten per cent larger than the actual value for 2001.

The first REPORT of the budget variable produces the following.

```
PRODUCT: OUTERWEAR - MEN
               --BUDGET--
               ---LINE---
TIME            REVENUE
-------------- ----------
YEAR02          1,100,000
Q1.02             275,000
Q2.02             225,000
Q3.02             200,000
Q4.02             300,000
JAN02             100,000
FEB02              90,000
MAR02              85,000
APR02              82,000
MAY02              70,000
JUN02              73,000
JUL02              64,000
```

```
AUG02              69,000
SEP02              67,000
OCT02              85,000
NOV02             105,000
DEC02             110,000
```

Example 9–4 then defines an aggregation map and adds contents to it with the ALLOCMAP command. The contents are a single RELATION command that specifies the PROPORTIONAL operator. The ALLOCATE command allocates the data from the YEAR02 parent down the hierarchy specified by the timeparent relation.

The REPORT of the budget variable after the first allocation produces the following.

```
PRODUCT: OUTERWEAR - MEN
               --BUDGET--
               ---LINE---
TIME            REVENUE
-------------- ----------
YEAR02          1,100,000
Q1.02             302,500
Q2.02             247,500
Q3.02             220,000
Q4.02             330,000
JAN02             110,000
FEB02              99,000
MAR02              93,500
APR02              90,200
MAY02              77,000
JUN02              80,300
JUL02              70,400
AUG02              75,900
SEP02              73,700
OCT02              93,500
NOV02             115,500
DEC02             121,000
```

The actual data for the first quarter of 2002 is assigned to the actual variable and then copied to the budget variable. The timelockvs valueset is defined and limited to the single value Q1.02.

A variable for a fileunit value is defined and is assigned the value returned by the FILEOPEN function. The CONSIDER and ALLOCMAP commands change the contents of the aggregation map so that the RELATION command includes the PROTECT argument.

The second ALLOCATE command allocates the data from the YEAR02 parent down the hierarchy specified by the timeparent relation but this allocation first subtracts the locked value for Q1.02 from the source value before distributing the remaining value. The command also sends error or informational messages to the allocerrlog file, which is specified by the errlogfunit fileunit.

The contents of the allocerrlog file are the following.

```
Dim      Source   Basis
TIME     BUDGET   BUDGET   Description
-------- -------- -------- -----------
  YEAR02   850000  1100000 Renormalizing data (6)
```

The source value for the allocation after normalization is 850,000 instead of the original value of 1,100,000. The REPORT of the budget variable after the second allocation, with the Q1.02 value protected, produces the following.

```
PRODUCT: OUTERWEAR - MEN
               --BUDGET--
               ---LINE---
TIME            REVENUE
-------------- ----------
YEAR02          1,100,000
Q1.02             250,000
Q2.02             263,793
Q3.02             234,483
Q4.02             351,724
JAN02              90,000
FEB02              82,000
MAR02              78,000
APR02              96,138
MAY02              82,069
JUN02              85,586
JUL02              75,034
AUG02              80,897
SEP02              78,552
OCT02              99,655
NOV02             123,103
DEC02             128,966
```

Example 9–4 Using the PROPORTIONAL Operator with the PROTECT Argument

```
LIMIT line TO 'REVENUE'
LIMIT product TO 'OUTERWEAR - MEN'
LIMIT time TO 'YEAR02' TO 'DEC02'
```

```
" Specify no decimal places
DECIMALS = 0

budget(time 'YEAR02') = actual(time 'YEAR01') * 1.1

REPORT DOWN time budget

DEFINE budgalloc AGGMAP
ALLOCMAP 'RELATION timeparent OPERATOR PROPORTIONAL'
ALLOCATE budget USING budgalloc

REPORT DOWN time budget

" Assign actual values for first quarter of 2002.
actual(time 'Q1.02' line 'REVENUE' product 'OUTERWEAR - MEN') = 250000
actual(time 'JAN02' line 'REVENUE' product 'OUTERWEAR - MEN') = 90000
actual(time 'FEB02' line 'REVENUE' product 'OUTERWEAR - MEN') = 82000
actual(time 'MAR02' line 'REVENUE' product 'OUTERWEAR - MEN') = 78000

LIMIT time TO 'Q1.02' 'JAN02' 'FEB02' 'MAR02'

" Copy the actual values to the budget variable
budget = actual

LIMIT time TO 'Q4.01' 'JAN02' 'FEB02' 'MAR02' 'Q1.02'

DEFINE timelockvs valueset time
LIMIT timelockvs TO 'Q1.02'

DEFINE errlogfunit VARIABLE INTEGER
errlogfunit = FILEOPEN('allocerrlog' WRITE)

CONSIDER budgalloc
ALLOCMAP 'RELATION timeparent OPERATOR PROPORTIONAL ARGS PROTECT timelockvs'
ALLOCATE budget USING budgalloc ERRORLOG errlogfunit

REPORT DOWN time budget
```

Part III

Analytic Workspace Management

Part III provides information about acquiring and generating data for an analytic workspace

It contains the following chapters:

- Chapter 10, "Working with Relational Tables"
- Chapter 11, "Reading Data from Files"
- Chapter 12, "Aggregating Data"

10

Working with Relational Tables

In this chapter, you will learn how to write OLAP DML programs that use the OLAP DML `SQL` command. With these programs, you can update relational tables and copy data between relational tables and analytic workspace objects.

This chapter includes the following topics:

- Issuing SQL Statements Through the OLAP DML
- Creating an Analytic Workspace from Relational Tables
- Example: Creating an Analytic Workspace from Sales History Tables
- Writing Data from Analytic Workspace Objects into Relational Tables
- Using Stored Procedures and Triggers
- Checking for Errors

Issuing SQL Statements Through the OLAP DML

SQL consists of statements that retrieve, delete, insert, change, and manipulate data stored in relational tables. You can embed SQL statements in OLAP DML programs using the OLAP DML `SQL` command show below.

```
SQL sql_statement
```

When formatting an SQL statement that is an argument to the OLAP DML `SQL` command, wherever you would normally use double quotes (`"`) in a SQL statement, use a single quote (`'`). In the OLAP DML, a double quote (`"`) indicates the beginning of a comment.

Supported SQL Statements

You can use almost any SQL statement that is supported by Oracle in the OLAP DML `SQL` command. You can use the `INSERT` command to copy data from analytic workspace objects into relational tables. You can use `FETCH` to copy data from relational tables into analytic workspace objects.

The following Oracle SQL extensions are also supported:

- The `FOR UPDATE` clause in the `SELECT` statement is supported in a cursor declaration so that you can update or delete data associated with the cursor.
- The `WHERE CURRENT OF` *`cursor`* clause is supported in `UPDATE` and `DELETE` statements for interactive modifications to a table.

Support is also provided for stored procedures and triggers. Using stored procedures is discussed in "Using Stored Procedures and Triggers" on page 10-32.

Unsupported SQL Statements

Ordinarily, you use the `SQL` command in an OLAP DML program, but you can also execute some SQL commands interactively in the OLAP Worksheet. When using SQL interactively, you would typically execute a `SELECT` command to produce a relational table of data. However, when using SQL within the OLAP DML, you must define a cursor which contains the `SELECT` statement as described in "Declaring a Cursor" on page 10-5.

Also, if you code `COMMIT` or `ROLLBACK` as arguments to the OLAP DML `SQL` command, the commands are ignored. You cannot rollback using the OLAP DML. To commit your changes, issue the OLAP DML `COMMIT` command.

Creating an Analytic Workspace from Relational Tables

When relational tables have been defined to the OLAP catalog using CWM1 metadata, you can use a tool provided with Oracle OLAP to design and populate an analytic workspace for the tables. For more information on creating an analytic workspace from relational tables in this manner, see *Oracle9i OLAP User's Guide*.

In other cases, you can design and populate an analytic workspace by taking the following steps:

1. Design the analytic workspace as described in "Process: Designing and Defining an Analytic Workspace to Hold Relational Data" on page 10-3.
   a. Define the analytic workspace using OLAP `AW CREATE` command.
   b. Define the analytic workspace objects using the OLAP `DEFINE` command.
2. Define, write, and execute OLAP DML programs to populate the analytic workspace objects with relational data as described in "Process: Writing Programs that Populate Analytic Workspaces with Relational Data" on page 10-4.
3. Aggregate the fact data up any hierarchies as described in Chapter 12, "Aggregating Data".

Process: Designing and Defining an Analytic Workspace to Hold Relational Data

One way that you can map a relational database to an analytic workspace is to take the following steps:

1. Identify the table columns that contain the fact data that you want to analyze. When the relational database is a data warehouse, these columns will be columns of a measure tables.
2. Identify the primary keys to the tables identified in step 1 and determine if any of these keys participate in any hierarchies. When the relational database is fully normalized, you can do this by following the foreign keys of the table. When the relational database contains summarized data, you can do this by, first, determining if the primary key columns are "children" of other columns, and then, following the "parent" columns up until you determine the complete hierarchy.

3. When there are hierarchies, decide if your applications need aggregated (summarized) fact data for each level of the hierarchy.
4. When your applications do *not* need aggregated data for any of the levels, then define a non-hierarchical dimension that you can use to hold the values of the primary key column as described in "Defining Dimensions" on page 3-8.
5. When your applications need aggregated fact data for some or all of the levels, then define the following analytic workspace objects to represent the hierarchy:
   a. An analytic workspace dimensions to hold the values of the levels for which you want aggregated data. You can define a hierarchical dimension as described in "Defining Hierarchical Dimensions and Variables That Use Them" on page 3-22; or you can define a concat dimension as described in "Defining Concat Dimensions and Variables That Use Them" on page 3-25.
   b. A self-relation for the hierarchy. This relation is dimensioned by the dimension described in step 5a. The values of this self-relation are the parents of each value in the hierarchy. For an example of a self-relation, see "Example: Self-relation" on page 3-15.
6. Define the variables for facts you identified in step 1 and for dimension attributes that you want to use in your analysis. Typically, these variables are dimensioned by the dimensions that you identified in steps 4 and 5. However, if any of these variables are sparsely populated, then you can define a composite for the dimensions, and dimension the variables by that composite.

For an example, of an analytic workspace designed following this process, see "Designing and Defining an Analytic Workspace for Sales History Data" on page 10-15.

Process: Writing Programs that Populate Analytic Workspaces with Relational Data

To populate the analytic workspace structures with data from relational tables, you write and execute one or more OLAP SQL programs that perform the following actions:

1. Define a SQL cursor and associate it with a `SELECT` statement or procedure as described in "Declaring a Cursor" on page 10-5.
2. Open the SQL cursor defined in step 1 as described in "Opening a Cursor" on page 10-8.
3. Retrieve and process data specified by the cursor opened in step 2 using wither the OLAP DML `SQL IMPORT` or `SQL FETCH` command as described in

"Importing and Fetching Relational Table Data into Analytic Workspace Objects" on page 10-8.

Note: You must declare and open a cursor from within a single OLAP DML program. You can fetch the data and close the cursor either in the same program or a different program.

4. Close the SQL cursor opened in step 2 as described in "Closing a Cursor" on page 10-13.
5. Cancel all SQL cursor definitions and free the memory resources of SQL cursors as described in "Cleaning up the SQL Cursors" on page 10-14.

Once the analytic workspace objects are populated, you can make these changes permanent using the OLAP DML `UPDATE` and `COMMIT` commands.

The rest of the topics in this section describe these steps in more detail. For examples of programs that populate an analytic workspace with data from relational tables, see "Populating Analytic Workspace Objects with Sales History Data" on page 10-19.

Declaring a Cursor

In an OLAP DML program, you cannot issue a `SELECT` statement interactively. Instead, you must define a cursor which contains the `SELECT` statement. In the context of a query, a cursor can be thought of as simply a row marker in a relational table of data resulting from a query. Instead of receiving the results of a query all at once, your program receives the results row by row using the cursor.

A `DECLARE CURSOR` statement associates a cursor by name with the results of a data query. As an argument to the OLAP DML `SQL` command, the `DECLARE CURSOR` statement has the following syntax.

```
SQL DECLARE cursor-name CURSOR FOR select-statement
```

Tip: You should write down `SELECT` statements that you think will retrieve the data you want to fetch. When possible, use an interactive interface such as SQL*Plus, SQL Worksheet, or OLAP Worksheet to test these SQL statements and make sure that they produce the results you expect. Afterward, you can modify these `SELECT` statements for use in your OLAP DML programs.

Example: Declaring a Cursor

In Example 10–1, "Declaring a Cursor", the cursor declaration selects rows from a relational table named costs in the sample Sales History (sh) schema. The costs table has several columns, including a column for product identification codes (prod_id) and a column for unit_price. The unit_price column is used in a WHERE clause to limit the returned rows to only those products in which the unit price is greater than $20.00.

Example 10–1 Declaring a Cursor

```
SQL DECLARE highprice CURSOR FOR -
   SELECT prod_id FROM costs -
      WHERE unit_price > 20
```

Using Variables in the WHERE Clause of the SELECT Statement

When you are declaring a cursor to be used by the OLAP DML SQL IMPORT command, you can only use literal values in the WHERE clause of a SELECT statement. However, when you are declaring a cursor to be used by the OLAP DML SQL FETCH command, you can use the values of input host variables instead of providing literal values in the WHERE clause of a SELECT statement.

Input host variables are values supplied by Oracle OLAP as parameters to a SQL command. They specify the data to be selected or provide values for data that is being modified. If you specify a dimension or a dimensioned variable, the first value in status is used; no implicit looping occurs, although you can use a FOR command to loop through all of the values. An input host variable can be any expression with an appropriate data type. When you use input host variables in a WHERE clause to match the data in a relational table, any required conversions between data types is performed wherever conversion is possible. The value of an input host variable is taken when a cursor is opened, not when it is declared.

An input host variable can be any expression preceded by a colon (for example, :myvar). However, if you specify a multidimensional expression, such as a variable or dimension, then the first value in status is used. Table 10–1 gives examples of expressions that can be used as input host variables. Example 10–2, "Using Input Host Variables" shows a program fragment that modifies the SQL command shown previously. Instead of using an explicit value in the WHERE clause, it uses the value of a local variable named set_price.

Table 10–1 Examples of Expressions That Can Be Used as Input Host Variables

Type of Expression	Example
Variable (database or local)	`:set_price`
Dimension	`:prod`
Qualified data reference	`:units(prod 'P8', geog 'G12', time 'T36')`
Program argument	`:newval`
Text expression	`:joinchars('first_name' 'last_name')`
Arithmetic expression	`:intpart(6.3049) + 1`
User-defined function	`:getgeog`

Example 10–2 Using Input Host Variables

```
VARIABLE set_price SHORT
set_price = 20
SQL DECLARE highprice CURSOR FOR -
   SELECT prod_id FROM costs -
      WHERE unit_price > :set_price
```

Using Conjunctions in a WHERE Clause

Because both the OLAP DML and SQL include `AND` and `OR` as part of their language syntax, you must use parentheses when using one of these conjunctions with an input host variable. Otherwise, the command might be ambiguous and produce unexpected results. Place the parentheses around the input host variable preceding `AND` and `OR`.

If a host variable expression begins with a parenthesis, then the matching right parenthesis is interpreted as the end of the host variable expression. If you plan to add more text to the expression after a matching right parenthesis, then you must enclose the entire expression with an extra set of parentheses.

The fragment of the program shown in Example 10–3 uses the values of two arguments to limit the range of values selected for the `prod_id` column of the relational table named `products`.

Example 10–3 **Using Conjunctions in a WHERE Clause**

```
prod1 = 415
prod1 = 49990
      ...
SQL DECLARE twoprods CURSOR FOR -
   SELECT prod_id FROM products -
      WHERE prod_id EQ :(prod1) -
      AND :prod2
```

Opening a Cursor

After the `SQL DECLARE CURSOR` command has associated a cursor with a selection of data, you use the `SQL OPEN` statement to get ready to retrieve the data. These commands for a particular cursor must appear in the same OLAP DML program and can *not* contain ampersand substitution.

The following is the syntax of the SQL command with an `OPEN` statement as an argument.

```
SQL OPEN cursor-name
```

The `SQL OPEN` command:

- Evaluates the input host variables (if any) used in the definition of the specified cursor.
- Determines the active set of the cursor (that is, the rows that satisfy the `SELECT` statement).
- Leaves the cursor in the open state for use by `SQL FETCH` or `SQL IMPORT`. The cursor is positioned before the first row of the result set

The active set of a cursor is determined when it is opened, and it is not updated later. Therefore, changing the value of an input host variable after opening its cursor does not affect the active set of a cursor.

Importing and Fetching Relational Table Data into Analytic Workspace Objects

After you open a cursor, you can use a `SQL IMPORT` or a `SQL FETCH` command statement to copy data from relational tables into analytic workspace objects. Before you use these SQL commands, ensure that you have access rights to the tables that you want to use.

`SQL IMPORT` or a `SQL FETCH` both copy data from relational tables into analytic workspace objects. Although `SQL FETCH` offers the most functionality, `SQL IMPORT`

offers improved performance when copying large amounts of data from relational tables into analytic workspace objects.

- **SQL FETCH** retrieves and processes data specified by a SQL cursor and assigns the retrieved data to OLAP objects. When you use a FETCH statement to retrieve data from relational tables, you must include it in a loop or use the LOOP argument to retrieve all of the rows of the active set of a cursor. Also, if you include a THEN clause, SQL FETCH may perform processing on the retrieved data. The following is the syntax of the SQL command using a FETCH statement as an argument.

```
SQL FETCH cursor [LOOP [loopcount]] INTO :targets... -
[THEN action-statements...]
```

- **SQL IMPORT** advances the cursor position to each subsequent row of the active set of a cursor and delivers the selected fields into analytic workspace objects. The following is the syntax of the OLAP DML SQL command using an IMPORT statement as an argument.

```
SQL IMPORT cursor INTO :targets...
```

In the syntax for SQL IMPORT and SQL FETCH, *targets* represents output host variables. An output host variable is an analytic workspace object that will be used to store the data retrieved from the relational tables. The order of the output host variables must match the order of the columns in the DECLARE CURSOR statement, and a colon must precede each output host variable name. The variable or dimension receiving the data must be defined already. It must also have a compatible data type.

For both IMPORT and FETCH, output host variables can be one or more of the following:

```
[MATCH] dimension|surrogate
APPEND  dimension
ASSIGN  surrogate
variable|qualified data reference|relation|composite
```

When an output host variable is a dimension, retrieved values are handled based on the keyword that you specify before the host variable name. You can specify either the MATCH keyword (the default) or the APPEND keyword.

- With the MATCH keyword, only values that are the same as existing values of the dimension are fetched, and an error is signalled when a new value is encountered. You use it when fetching data into a variable whose dimensions

are already maintained; the dimensions are included in the fetch only to align the data.

- With the `APPEND` keyword, all values that do not match are added to the end of the list of dimension values. Also, for `FETCH`, values can be appended to an output host variable based on position using the following syntax for target:

  ```
  APPEND [position] dimension
  ```

Table 10–2 provides examples of expressions that can be used as output host variables.

Table 10–2 Examples of Expressions That Can Be Used as Output Host Variables

Type of Expression	Example
Variable (database or local)	`:sales_quantity_sold`
Dimension or surrogate	`:prodid`
Qualified data reference	`:sales_quantity_sold(prod_id 415 cust_id 18670 time_id '1998-01-04' channel_id 'S' promo_id 9999)`

Whenever you fetch data into a dimensioned workspace variable, you must include the dimension values in the fetch. While you can add new dimension values at the same time, you do not need to add them when they already exist in your analytic workspace; instead, you use the dimension values in the fetch to align the data. In either case, be sure to fetch the dimension values before you fetch the values of the variable. Otherwise, the fetch will not loop through the dimension value.

Important: When data is written into a dimension, it temporarily limits the status of the dimension to the value being matched or appended. This means that when the `IMPORT` statement or the `FETCH` statement also includes output host variables that are dimensioned by the specified dimension, the temporary status is observed when values are assigned to those variables.

Null values in a relational table are equivalent to `NA`s. In OLAP DML variables, null values do not pose a problem; they appear as `NA`s. However, you cannot have a dimension value of `NA`. Therefore, any rows that have a value of `null` are discarded in a column being fetched into a dimension.

Example: Copying Relational Table Data into Analytic Workspace Objects

Sometimes you want to copy data from relational tables into the analytic workspace to perform a quick analysis. For example, the sample Sales History database includes the `sales` table (described in Example 10–4 on page 10-12) whose keys are `prod_id`, `cust_id`, `time_id`, `channel_id`, and `promo_id` and that contains two facts (`quantity_sold` and `amount_sold`).

Assume that you want to forecast the quantity sold for product 415 for the year 2002 using the forecasting commands available in the OLAP DML. In order to perform this analysis using the OLAP DML, the data must be in an analytic workspace. To copy the data into the analytic workspace, you must define the analytic workspace objects to hold the data, write an OLAP DML program to copy the data from the relational table to the analytic workspace objects, and, then, execute that program.

The simplest way to map the `sales` table to analytic workspace objects is to define one analytic workspace dimension for each of the key columns (`aw_prod_id`, `aw_cust_id`, `aw_time_id`, `aw_channel_id`, and `aw_promo_id`) and to define analytic workspace variables (dimensioned by those dimensions) to hold the data from the other columns (`aw_quantity_sold` and `aw_amount_sold`). However, in this case, the variables will be quite sparse along the time dimension. To avoid this sparsity, you can define a composite that represents all of the key dimensions and define the analytic workspace variables using this composite as shown in Example 10–5, "Analytic Workspace Definitions for Sales Data" on page 10-12.

Example 10–6, "import_sales_for_prod415 Program" on page 10-12) illustrates using `SQL IMPORT` to copy the data from the relational table into the analytic workspace objects. The `fetch_sales_for_prod415` program (shown in Example 10–7, "fetch_sales_for_prod415 Program" on page 10-13) illustrates using `SQL FETCH` to copy the data from the relational table into the analytic workspace objects. Both of these programs assume that values for `aw_prod_id`, `aw_cust_id`, `aw_time_id`, `aw_channel_id`, and `aw_promo_id` have not previously been copied into the analytic workspace. When you have defined a composite, Oracle OLAP automatically populates the composite as it populates the other analytic workspace objects.

Example 10–4 Description of the sales Table

```
PROD_ID                                     NOT NULL NUMBER(6)
CUST_ID                                     NOT NULL NUMBER
TIME_ID                        NOT NULL DATE
CHANNEL_ID                                  NOT NULL CHAR(1)
PROMO_ID                                    NOT NULL NUMBER(6)
QUANTITY_SOLD                               NOT NULL NUMBER(3)
AMOUNT_SOLD                                 NOT NULL NUMBER(10,2)
```

Example 10–5 Analytic Workspace Definitions for Sales Data

```
DEFINE aw_prod_id DIMENSION NUMBER (6)
DEFINE aw_cust_id DIMENSION NUMBER (6)
DEFINE aw_date DIMENSION TEXT
DEFINE aw_channel_id DIMENSION TEXT
DEFINE aw_promo_id DIMENSION NUMBER (6)
DEFINE aw_sales_dims COMPOSITE <aw_prod_id aw_cust_id aw_date -
     aw_channel_id aw_promo_id>
DEFINE aw_sales_quantity_sold VARIABLE NUMBER (3) <aw_sales_dims <aw_prod_id -
     aw_cust_id aw_date aw_channel_id paw_romo_id>>
DEFINE aw_sales_amount_sold VARIABLE NUMBER (10,2) <aw_sales_dims <aw_prod_id -
     aw_cust_id aw_date aw_channel_id aw_promo_id>>
```

Example 10–6 import_sales_for_prod415 Program

```
ALLSTAT
NLS_DATE_FORMAT = '<YYYY><MM><DD>'
DATEFORMAT = '<YYYY>-<MM>-<DD>'
" Declare a cursor named GRABDATA
SQL DECLARE grabdata CURSOR FOR SELECT prod_id, cust_id, time_id, -
    channel_id, promo_id, quantity_sold, amount_sold FROM sh.sales -
    WHERE prod_id = 415
" Import new values into the analytic workspace objects
SQL IMPORT grabdata INTO :APPEND aw_prod_id -
                         :APPEND aw_cust_id -
                         :APPEND aw_date -
                         :APPEND aw_channel_id -
                         :APPEND aw_promo_id -
                         :aw_sales_quantity_sold -
                         :aw_sales_amount_sold
" Update the analytic workspace and make the updates permanent
UPDATE
COMMIT
```

Example 10–7 fetch_sales_for_prod415 Program

```
ALLSTAT
NLS_DATE_FORMAT = '<YYYY><MM><DD>'
DATEFORMAT = '<YYYY>-<MM>-<DD>'
" Declare a cursor named GRABDATA
SQL DECLARE grabdata CURSOR FOR SELECT prod_id, cust_id, time_id, -
    channel_id, promo_id, quantity_sold, amount_sold FROM sh.sales -
    WHERE prod_id = 415
" Open the cursor
SQL OPEN grabdata
" Fetch new values into the analytic workspace objects
SQL FETCH grabdata LOOP INTO :APPEND aw_prod_id -
                             :APPEND aw_cust_id -
                             :APPEND aw_date -
                             :APPEND aw_channel_id -
                             :APPEND aw_promo_id -
                             :aw_sales_quantity_sold -
                             :aw_sales_amount_sold
" Close the cursor
SQL CLOSE grabdata
" Cleanup from SQL query
SQL CLEANUP
" Update the analytic workspace and make the updates permanent
UPDATE
COMMIT
```

Closing a Cursor

After you have used a cursor to retrieve all the data in its active set, you close the cursor. If you want to use the cursor again to retrieve data starting from the first row of its active set, then you can use the `OPEN` statement without having to declare the cursor again. The `CLOSE` statement does not cancel a cursor declaration; it only renders the active set undefined.

The following is the syntax of the `CLOSE` statement when it is used as an argument in the OLAP DML `SQL` command.

```
SQL CLOSE cursor-name
```

Cleaning up the SQL Cursors

Once you are completely done making OLAP DML SQL calls, you should cancel all the SQL cursor declarations and free the memory resources for all SQL cursors. You perform these actions by using `CLEANUP` as the argument to the OLAP DML `SQL` command:

```
SQL CLEANUP
```

After you have cancelled all SQL cursors in this manner, you cannot use them again unless you issue new `SQL DECLARE CURSOR` and `SQL OPEN` commands.

Example: Creating an Analytic Workspace from Sales History Tables

The sample Sales History database, which is fully described in *Oracle9i Sample Schemas*, has six dimension tables and two fact tables:

- `countries`, a dimension table that has a primary key of `country_id`.
- `customers`, a dimension table that has a primary key of `customer_id` and a foreign key of `country_id`.
- `promotions`, a dimension table that has a primary key of `promo_id`.
- `products`, a dimension table that has a primary key of `product_id`.
- `channels`, a dimension table that has a primary key of `channel_id`.
- `times`, a dimension table that has a primary key of `time_id`.
- `sales`, a fact table that has `customer_id`, `promo_id`, `product_id`, `channel_id`, and `time_id` as keys.
- `costs`, a fact table that has which has `product_id` and `time_id` as keys.

Assume that you want to analyze all of the fact data in the sample Sales History database. In order to do this you need to design and define an analytic workspace as described in "Designing and Defining an Analytic Workspace for Sales History Data" on page 10-15. Then you need to write OLAP DML programs to copy the necessary relational data into the analytic workspace as described in "Populating Analytic Workspace Objects with Sales History Data" on page 10-15.

Designing and Defining an Analytic Workspace for Sales History Data

The analytic workspace for Sales History was designed and defined following the process described in "Process: Designing and Defining an Analytic Workspace to Hold Relational Data" on page 10-3. The actual steps are outlined below:

1. An analytic workspace named awsh was created using the following OLAP DML command.

   ```
   AW CREATE awsh
   ```

2. The fact data was identified. In the sales table, the quantity_sold and the amount_sold columns were identified as containing facts for analysis. While, in the costs table, the unit_cost and unit_price columns contain fact data of interest.
3. The primary keys to the sales and costs tables were identified. The primary keys of sales are prod_id, cust_id, time_id, channel_id, and promo_id. The primary keys of costs are prod_id and time_id.
4. Looking at the primary keys, the following hierarchies in the Sales History database were identified:
   - Products — This hierarchy has four levels (prod_id, prod_subcategory, prod_category, and products_all) that map to columns in the products tables. The lowest level of the hierarchy is prod_id and the highest level is products_all.
   - Channels — This hierarchy has three levels (channel_id, channel_class, and channels_all) that map to columns in the channels tables. The lowest level of the hierarchy is channel_id and the highest level is channels_all.
   - Promotions — This hierarchy has four levels (promo_id, promo_subcategory, promo_category, and promos_all) that map to columns in the promotions tables. The lowest level of the hierarchy is promo_id and the highest level is promos_all.
   - Customers — This hierarchy has seven levels that map to columns in two different relational tables. Four of these levels (country_id, region, subreagion, and world) map to columns in the countries table and three levels (cust_id, state_province, and city) map to columns in the customers table. The lowest level of the hierarchy is cust_id and the highest level is world.

- Time hierarchies— Two time hierarchies were identified: Calendar and Fiscal.

  Calendar Time— This hierarchy has five levels (`time_id`, `cal_week_num`, `cal_month_num`, `cal_quarter_num`, and `cal_year`) that map to columns in the `times` table.

  Fiscal Time — This hierarchy has five levels (`time_id`, `fis_week_num`, `fis_month_num`, `fis_quarter_num`, and `fis_year`) that map to columns in the `times` table.

Also, a one-to-many relationship between `prod_id` and `supplier_id` was identified.

5. Our application needs to aggregate (summarize) fact data for each level of the Products, Customers, Channels, and Promotions hierarchies. For the time hierarchies, our application only needs hierarchies with two levels — the lowest level of the hierarchy (`time_id`) and year (`cal_year` and `fis_year`).

6. The following analytic workspace objects were defined to represent the hierarchies:

   - For Products, Customers, Channels, and Promotions hierarchies, a dimension was defined for each level of the hierarchy, a concat dimension was defined for each hierarchy, and a child-parent self-relation was defined for each concat dimension. These definitions are shown in examples Example 10–8 on page 10-17 through Example 10–11 on page 10-18.

   - For the time hierarchies, two hierarchies were defined. A dimension containing the names of the two hierarchies was created. Base dimensions were defined for `time_id`, `fis_year`, and `cal_year` and a concat dimension was defined that specified all of these dimensions as base dimensions. Since there are two time hierarchies the child-parent self-relation created for the Times hierarchy is dimensioned by both the concat dimension and the hierarchies (by name). These definitions are shown in Example 10–12, "Analytic Workspace Definitions for the Times Hierarchies" on page 10-18

   - For the facts (`quantity_sold`, `amount_sold`, the `unit_cost` and `unit_price`), analytic workspace variables were defined. All of these variables would be sparsely populated if they were dimensioned by the concat dimensions, so one composite was defined for each variable. The variables are dimensioned by those composites. The definitions for the variables for the fact data is shown in Example 10–13, "Analytic Workspace

Definitions for Variables for Facts" on page 10-18 include definitions for these composites.

Our applications had no need of other data. However, Example 10–14, "Definitions for Variables for Promotions Dimension Attributes" on page 10-19 show definitions of analytic workspace variables to which promotions attributes could be mapped. For an example of how to define relational views of the `awsh` analytic workspace see the example of using the `OLAP_TABLE` function in *Oracle9i OLAP User's Guide*.

Example 10–8 Analytic Workspace Definitions for the Products Hierarchy

```
DEFINE aw_prod_id DIMENSION NUMBER (6)
DEFINE aw_prod_subcategory DIMENSION TEXT
DEFINE aw_prod_category DIMENSION TEXT
DEFINE aw_products_all DIMENSION TEXT
DEFINE aw_products DIMENSION CONCAT (aw_products_all -
                                  aw_prod_category -
                                  aw_prod_subcategory -
                                  aw_prod_id)
DEFINE aw_products.parents RELATION aw_products <aw_products>
DEFINE aw_supplier_id DIMENSION TEXT
DEFINE aw_prod_id.aw_supplier_id RELATION aw_supplier_id <aw_prod_id>
```

Example 10–9 Analytic Workspace Definitions for the Channels Hierarchy

```
DEFINE aw_channel_id DIMENSION TEXT
DEFINE aw_channel_class DIMENSION TEXT
DEFINE aw_channels_all DIMENSION TEXT
DEFINE aw_channels DIMENSION CONCAT(aw_channels_all -
                                 aw_channel_class -
                                 aw_channel_id)
DEFINE aw_channels.parents RELATION aw_channels <aw_channels>
```

Example 10–10 Analytic Workspace Definitions for the Promotions Hierarchy

```
DEFINE aw_promo_id DIMENSION NUMBER(6)
DEFINE aw_promo_subcategory DIMENSION TEXT
DEFINE aw_promo_category DIMENSION TEXT
DEFINE aw_promos_all DIMENSION TEXT
DEFINE aw_promos DIMENSION CONCAT(aw_promos_all -
                               aw_promo_category -
                               aw_promo_subcategory -
                               aw_promo_id)
DEFINE aw_promos.parents RELATION aw_promos <aw_promos>
```

Example 10–11 Analytic Workspace Definitions for the Customers Hierarchy

```
DEFINE aw_cust_id DIMENSION NUMBER (8)
DEFINE aw_city DIMENSION TEXT
DEFINE aw_state_province DIMENSION TEXT
DEFINE aw_country_id DIMENSION TEXT
DEFINE aw_subregion DIMENSION TEXT
DEFINE aw_region DIMENSION TEXT
DEFINE aw_world DIMENSION TEXT
DEFINE aw_customers DIMENSION CONCAT(aw_world -
                                  aw_region -
                                  aw_subregion -
                                  aw_country_id -
                                  aw_state_province -
                                  aw_city -
                                  aw_cust_id)
DEFINE aw_customers.parents RELATION aw_customers <aw_customers>
```

Example 10–12 Analytic Workspace Definitions for the Times Hierarchies

```
DEFINE aw_time_id DIMENSION TEXT
DEFINE aw_cal_year DIMENSION NUMBER(4)
DEFINE aw_fis_year DIMENSION NUMBER(4)
DEFINE aw_times DIMENSION CONCAT (aw_cal_year -
                                  aw_fis_year -
                                  aw_time_id)
DEFINE aw_times_hiernames DIMENSION TEXT
DEFINE aw_times.parents RELATION aw_times <aw_times aw_times_hiernames>
```

Example 10–13 Analytic Workspace Definitions for Variables for Facts

```
DEFINE aw_costsdims COMPOSITE <aw_products aw_times>
DEFINE aw_unit_cost VARIABLE  NUMBER (10,2) <aw_costsdims -
        <aw_products aw_times>>
DEFINE aw_unit_price VARIABLE NUMBER (10,2) <aw_costsdims -
        <aw_products aw_times>>

DEFINE aw_salesdims COMPOSITE <aw_products aw_customers aw_times -
               aw_channels aw_promos>
DEFINE aw_quantity_sold VARIABLE NUMBER(3) <aw_salesdims -
              <aw_products aw_customers aw_times aw_channels aw_promos>>
DEFINE aw_amount_sold VARIABLE NUMBER(10,2) <aw_salesdims -
              <aw_products aw_customers aw_times aw_channels aw_promos>>
```

Example 10–14 Definitions for Variables for Promotions Dimension Attributes

```
DEFINE aw_promo_name VARIABLE TEXT <aw_promo_id>
DEFINE aw_promo_cost VARIABLE NUMBER(10,2) <aw_promo_id>
DEFINE aw_promo_begin_date VARIABLE DATE <aw_promo_id>
DEFINE aw_promo_end_date VARIABLE DATE <aw_promo_id>
```

Populating Analytic Workspace Objects with Sales History Data

In this example there are a number of OLAP DML programs that copy the data from the relational Sales History database into the objects in the analytic workspace named awsh:

- The following programs copy data from the relational tables into analytic workspace dimensions and variables:
  - Example 10–15, "get_products_hier Program" on page 10-20 copies the data from the dimension tables into the base dimensions of the aw_products concat dimension using SQL FETCH commands with the APPEND keyword. As the base dimensions of aw_products are populated, Oracle OLAP automatically populates aw_products, itself. As the THEN clause of the SQL FETCH command executes, Oracle OLAP fetches data into the child-parent self-relation for aw_products. This program also populates the aw_supplier_id dimension and its relation.
  - Example 10–16, "get_channels_hier Program" on page 10-21, Example 10–17, "get_promos_hier Program" on page 10-22, Example 10–18, "get_customers_hier Program" on page 10-22, and Example 10–19, "get_times_hiers Program" on page 10-24 copy the data from the dimension tables into analytic workspace dimensions and relations that are used to represent hierarchical dimensions. Because these dimensions are empty before these programs execute, the SQL FETCH command uses the APPEND keyword. As the base dimensions are populated, Oracle OLAP automatically populates the concat dimension that represents the hierarchy. As the THEN clause of the SQL FETCH command executes, Oracle OLAP fetches data into the child-parent self-relation for concat dimension that represents the hierarchy.
- The following programs copy the facts from the relational tables into analytic workspace variables. These examples assume that the base dimension for these variables are already populated. Consequently, the SQL FETCH commands in these programs use the MATCH keyword. Also, because the composite that the variables are dimensioned by is constructed of concat dimensions, the SQL FETCH commands uses a QDR to specify dimension values for the variable.

  - Example 10–20, "get_costs Program" on page 10-25 copies the facts from the `costs` table into analytic workspace variables.
  - Example 10–21, "get_sales Program" on page 10-26 copies the facts from the `sales` table into analytic workspace variables.
- Example 10–23, "get_promos_attr Program" on page 10-27 copies attribute data from the Promotions dimension table into analytic workspace variables. This program assumes that the base dimensions are already populated and uses a `SQL IMPORT` command with the `MATCH` keyword.

Example 10–15 get_products_hier Program

```
ALLSTAT
" Fetch values into the products hierarchy
SQL DECLARE grabprods CURSOR FOR SELECT prod_total, -
                                        prod_category, -
                                        prod_subcategory, -
                                        prod_id -
                                   FROM sh.products
SQL OPEN grabprods
SQL FETCH grabprods LOOP INTO :APPEND aw_products_all -
                              :APPEND aw_prod_category -
                              :APPEND aw_prod_subcategory -
                              :APPEND aw_prod_id

SQL CLOSE grabprods
SQL CLEANUP
" Update the analytic workspace and make the updates permanent
UPDATE
COMMIT
" Fetch values into supplier_id
SQL DECLARE grabsupid CURSOR FOR SELECT supplier_id -
                                 FROM sh.products
SQL OPEN grabsupid
SQL FETCH grabsupid LOOP INTO :APPEND aw_supplier_id
SQL CLOSE grabsupid
SQL CLEANUP
" Update the analytic workspace and make the updates permanent
UPDATE
COMMIT
```

```
" Populate self-relation for concat dimension
" and relation between aw_prod_id and aw_supplier_id
SQL DECLARE makerels CURSOR FOR SELECT prod_total, -
                                       prod_category, -
                                       prod_subcategory, -
                                       prod_id, -
                                       supplier_id -
                                 FROM sh.products
SQL OPEN makerels
SQL FETCH makerels LOOP INTO :MATCH aw_products_all -
                             :MATCH aw_prod_category -
                             :MATCH aw_prod_subcategory -
                             :MATCH aw_prod_id -
                             :MATCH aw_supplier_id -
            THEN aw_products.parents(aw_products aw_prod_id) -
                = aw_products(aw_prod_subcategory aw_prod_subcategory) -
            aw_products.parents(aw_products aw_prod_subcategory) -
               = aw_products(aw_prod_category aw_prod_category) -
            aw_products.parents(aw_products aw_prod_category) -
               = aw_products(aw_products_all aw_products_all) -
            aw_prod_id.aw_supplier_id = aw_supplier_id
SQL CLOSE makerels
SQL CLEANUP
" Update the analytic workspace and make the updates permanent
UPDATE
COMMIT
```

Example 10–16 get_channels_hier Program

```
ALLSTAT
" Fetch values for the Channels hierarchy
" and populate self-relation for the hierarchy
SQL DECLARE grabchanneldata CURSOR FOR SELECT channel_total, -
                                       channel_class, -
                                       channel_id -
                                FROM sh.channels
SQL OPEN grabchanneldata
" Fetch values into analytic workspace objects for the the channels hierararchy
SQL FETCH grabchanneldata LOOP INTO :APPEND aw_channels_all -
                             :APPEND aw_channel_class -
                             :APPEND aw_channel_id -
         THEN aw_channels.parents(aw_channels aw_channel_id) -
              = aw_channels(aw_channel_class aw_channel_class) -
         aw_channels.parents(aw_channels aw_channel_class) -
              = aw_channels(aw_channels_all aw_channels_all)
```

```
SQL CLOSE grabchanneldata
SQL CLEANUP
" Update the analytic workspace and make the updates permanent
UPDATE
COMMIT
```

Example 10–17 get_promos_hier Program

```
ALLSTAT
" Fetch values for the Promos hierarchy
" and populate self-relation for the hierarchy
SQL DECLARE grabpromodata CURSOR FOR SELECT promo_total, -
                                            promo_category, -
                                            promo_subcategory, -
                                            promo_id -
                                 FROM sh.promotions
SQL OPEN grabpromodata
SQL FETCH grabpromodata LOOP INTO :APPEND aw_promos_all -
                                  :APPEND aw_promo_category -
                                  :APPEND aw_promo_subcategory -
                                  :APPEND aw_promo_id -
        THEN aw_promos.parents(aw_promos aw_promo_id) -
                    = aw_promos(aw_promo_subcategory aw_promo_subcategory) -
                  aw_promos.parents(aw_promos aw_promo_subcategory) -
                   = aw_promos(aw_promo_category aw_promo_category) -
            aw_promos.parents(aw_promos aw_promo_category) -
             = aw_promos(aw_promos_all aw_promos_all)
SQL CLOSE grabpromodata
SQL CLEANUP
" Update the analytic workspace and make the updates permanent
UPDATE
COMMIT
```

Example 10–18 get_customers_hier Program

```
ALLSTAT
" Fetch values for the Customers hierarchy from the countries table
" and populate the self-relation for the hierarchy with these values
SQL DECLARE grabcountrydata CURSOR FOR SELECT country_total, -
                               country_region, -
                               country_subregion, -
                               country_id -
                           FROM sh.countries
```

```
SQL OPEN grabcountrydata
SQL FETCH grabcountrydata LOOP INTO :APPEND aw_world -
                                    :APPEND aw_region -
                                    :APPEND aw_subregion -
                                    :APPEND aw_country_id -
      THEN aw_customers.parents(aw_customers aw_country_id) = -
             aw_customers(aw_subregion aw_subregion) -
           aw_customers.parents(aw_customers aw_subregion) = -
             aw_customers(aw_region aw_region) -
           aw_customers.parents(aw_customers aw_region) = -
             aw_customers(aw_world aw_world)
SQL CLOSE grabcountrydata
SQL CLEANUP
" Update the analytic workspace and make the updates permanent
UPDATE
COMMIT
" Fetch values for the Customers hierarchy from the customers table
" and populate the self-relation for the hierarchy with these values
SQL DECLARE grabcustdata CURSOR FOR SELECT country_id, -
                                           cust_state_province, -
                                           cust_city, -
                                           cust_id -
                                 FROM sh.customers
SQL OPEN grabcustdata
SQL FETCH grabcustdata LOOP INTO :MATCH aw_country_id -
                                 :APPEND aw_state_province -
                                 :APPEND aw_city -
                                 :APPEND aw_cust_id -
                THEN aw_customers.parents(aw_customers aw_cust_id) = -
                   aw_customers(aw_city aw_city) -
               aw_customers.parents(aw_customers aw_city) = -
                   aw_customers(aw_state_province aw_state_province) -
               aw_customers.parents(aw_customers aw_state_province) = -
                   aw_customers(aw_country_id aw_country_id)
SQL CLOSE grabcustdata
SQL CLEANUP
" Update the analytic workspace and make the updates permanent
UPDATE
COMMIT
```

Example 10–19 get_times_hiers Program

```
NLS_DATE_FORMAT = '<YYYY><MM><DD>'
DATEFORMAT = '<YYYY>-<MM>-<DD>'
" Populate the hierachy name dimension with names of hierarchies
MAINTAIN aw_times_hiernames ADD 'Calendar' 'Fiscal'
" Update the analytic workspace and make the updates permanent
UPDATE
COMMIT

" Fetch values for the CalTimes and FisTimes hierarchies
" and populate self-relation time
SQL DECLARE grabcalyear CURSOR FOR SELECT calendar_year -
                                FROM sh.times
SQL OPEN grabcalyear
SQL FETCH grabcalyear LOOP INTO :APPEND aw_cal_year
SQL CLOSE grabcalyear
SQL CLEANUP
" Update the analytic workspace and make the updates permanent
UPDATE
COMMIT
SQL DECLARE grabfisyear CURSOR FOR SELECT fiscal_year -
                                    FROM sh.times
SQL OPEN grabfisyear
SQL FETCH grabfisyear LOOP INTO :APPEND aw_fis_year
SQL CLOSE grabfisyear
SQL CLEANUP
" Update the analytic workspace and make the updates permanent
UPDATE
COMMIT
SQL DECLARE grabtimeid CURSOR FOR SELECT time_id -
                                    FROM sh.times
SQL OPEN grabtimeid
SQL FETCH grabtimeid LOOP INTO :APPEND aw_time_id
SQL CLOSE grabtimeid
SQL CLEANUP
" Update the analytic workspace and make the updates permanent
UPDATE
COMMIT
ALLSTAT
LIMIT aw_times_hiernames TO 'Calendar'
SQL DECLARE makecalhier CURSOR FOR SELECT calendar_year, -
                                            time_id -
                                FROM sh.times
SQL OPEN makecalhier
```

```
SQL FETCH makecalhier LOOP INTO :MATCH aw_cal_year -
                                :MATCH aw_time_id -
       THEN aw_times.parents(aw_times aw_time_id) -
              = aw_times(aw_cal_year aw_cal_year)
SQL CLOSE makecalhier
SQL CLEANUP
" Update the analytic workspace and make the updates permanent
ALLSTAT
UPDATE
COMMIT
LIMIT aw_times_hiernames TO 'Fiscal'
SQL DECLARE makefishier CURSOR FOR SELECT fiscal_year, -
                                        time_id -
                                FROM sh.times
SQL OPEN makefishier
SQL FETCH makefishier LOOP INTO :MATCH aw_fis_year -
                                :MATCH aw_time_id -
       THEN aw_times.parents(aw_times aw_time_id) -
              = aw_times(aw_fis_year aw_fis_year)
SQL CLOSE makefishier
SQL CLEANUP
" Update the analytic workspace and make the updates permanent
ALLSTAT
UPDATE
COMMIT
```

Example 10–20 get_costs Program

```
ALLSTAT
NLS_DATE_FORMAT = '<YYYY><MM><DD>'
DATEFORMAT = '<YYYY>-<MM>-<DD>'
" Declare a cursor named grabcosts
SQL DECLARE grabcosts CURSOR FOR SELECT prod_id, -
                                        time_id, -
                                        unit_cost, -
                                        unit_price -
                            FROM sh.costs
" Open the cursor
SQL OPEN grabcosts
```

```
" Import the data
SQL FETCH grabcosts LOOP INTO :MATCH aw_prod_id -
                         :MATCH aw_time_id -
                         :aw_unit_cost (aw_products aw_prod_id -
                    aw_times aw_time_id) -
                         :aw_unit_price (aw_products aw_prod_id -
                    aw_times aw_time_id)
" Close the cursor
SQL CLOSE grabcosts
" Cleanup from SQL query
SQL CLEANUP
" Update and make changes permanent
UPDATE
COMMIT
```

Example 10–21 get_sales Program

```
ALLSTAT
NLS_DATE_FORMAT = '<YYYY><MM><DD>'
DATEFORMAT = '<YYYY>-<MM>-<DD>'
" Declare a cursor named grabsales
SQL DECLARE grabsales CURSOR FOR SELECT prod_id, -
                                        cust_id, -
                                        time_id, -
                                        channel_id, -
                                        promo_id, -
                                        quantity_sold, -
                                        amount_sold -
                      FROM sh.sales
```

```
" Open the cursor
SQL OPEN grabsales
" Import values into analytic workspace objects
SQL FETCH grabsales LOOP INTO :MATCH aw_prod_id -
                              :MATCH aw_cust_id -
                              :MATCH aw_time_id -
                              :MATCH aw_channel_id -
                              :MATCH aw_promo_id -
                              :aw_quantity_sold (aw_products aw_prod_id -
                                             aw_customers aw_cust_id -
                                              aw_times aw_time_id -
                                            aw_channels aw_channel_id -
                                              aw_promos aw_promo_id) -
                              :aw_amount_sold (aw_products aw_prod_id -
                                          aw_customers aw_cust_id -
                                          aw_times aw_time_id -
                                          aw_channels aw_channel_id -
                                           aw_promos aw_promo_id)
" Close the cursor
SQL CLOSE grabsales
" Cleanup from SQL query
SQL CLEANUP
" Update and make changes permanent
UPDATE
COMMIT
```

Example 10–22 Definitions for Variables for Promotions Dimension Attributes

```
DEFINE aw_promo_name VARIABLE TEXT <aw_promo_id>
DEFINE aw_promo_cost VARIABLE NUMBER(10,2) <aw_promo_id>
DEFINE aw_promo_begin_date VARIABLE DATE <aw_promo_id>
DEFINE paw_romo_end_date VARIABLE DATE <aw_promo_id>
```

Example 10–23 get_promos_attr Program

```
ALLSTAT
" Declare a cursor named grabpromoattr
SQL DECLARE grabpromoattr CURSOR FOR SELECT promo_id, -
                                            promo_name, -
                                            promo_cost, -
                                            promo_begin_date, -
                                            promo_end_date -
                        FROM sh.promotions
" Open the cursor
SQL OPEN grabpromoattr
```

```
" Import new values into the analytic workspace objects
SQL IMPORT grabpromoattr INTO :MATCH aw_promo_id -
                                    :aw_promo_name -
                                    :aw_promo_cost -
                                    :aw_promo_begin_date -
                                    :aw_promo_end_date
" Close the cursor
SQL CLOSE grabpromoattr
" Cleanup from SQL query
SQL CLEANUP
" Update and make changes permanent
UPDATE
COMMIT
```

Writing Data from Analytic Workspace Objects into Relational Tables

To copy data from analytic workspace object you can simply use the SQL `INSERT` or `UPDATE` statements as arguments to the OLAP DML `SQL` command. In this case, you code the OLAP DML `SQL` in a loop and you use the analytic workspace variables as input host variables in your SQL statements. However, you can improve performance by doing a direct insert using the `PREPARE` and `EXECUTE` statements as arguments to the OLAP DML command.

Tip: You can access data in an analytic workspace in a `SQL SELECT` statement without copying data from the analytic workspace into relational tables by defining a view of the analytic workspace data. For more information on defining relational views of analytic workspace data, see *Oracle9i OLAP User's Guide*.

Using SQL PREPARE and SQL EXECUTE

The syntax of the `PREPARE` and `EXECUTE` statements is shown below.

```
SQL PREPARE statement-name FROM sql-statement [insert-options]
SQL EXECUTE statement-name
```

The arguments for these statements are described below:

- *`statement-name`* is the name that you assign to the executable code produced from *`sql-statement`*. You can redefine *`statement-name`* just by issuing another `SQL PREPARE` command.
- *`sql-statement`* is the SQL statement that you want to precompile for more efficient execution. It cannot contain ampersand (`&`) substitution or variables that are undefined when the program is compiled.
- *`insert-options`* are `DIRECT`, `NOLOG`, and `PARTITION` that apply when *`sql-statement`* is an `INSERT` statement. When you prepare an `INSERT` statement and do not specify any values for the insert options, Oracle OLAP specifies `NO` for the `DIRECT` and `NOLOG` insert options and does not specify a value for the `PARTITION` option. Thus, by default, a prepared `INSERT` is a normal insert, redo information is recorded in the redo log files, and other sessions cannot insert data into the table into which your program is inserting values. You can improve performance of your `INSERT`, by changing the values of these options. You can specify that you want a direct insert, that you do not want `the` redo information recorded in the redo log files, and the partition or subpartition that you want locked (that is, the partition or subpartition into which you do not want another session to be able to insert data).

Performing a Direct Insert

Direct-path insert enhances performance during insert operations and is similar to the functionality of Oracle's direct-path loader utility, SQL*Loader. To specify a direct-path insert, specify `DIRECT=YES` as the first insert option in the OLAP DML `SQL PREPARE INSERT` command.

Inserting Workspace Data into Relational Tables: Example

Suppose that you have been using the OLAP DML to plan the introduction of a new product line, and now you want to add information about the product ids and the product names for these new products to the Sales History database. You can copy this information from your analytic workspace into the `products` table using an

OLAP DML program. The definitions for the analytic workspace objects that contain the data are shown in Example 10–24.

The program fragment in Example 10–25 shows how you would use a FOR loop so that all product values currently in status are copied to a table named Products. Example 10–25 will run much more efficiently when the INSERT statement is compiled with the PREPARE statement. Example 10–26 shows the PREPARE statement being used to compile the INSERT statement with a name of write_products, which is then run by an EXECUTE statement within the FOR loop.

Example 10–24 Analytic Workspace definitions for add_newprods program

```
DEFINE aw_prod_id DIMENSION NUMBER (10,0)
DEFINE aw_product_name DIMENSION TEXT
```

The program fragment in Example 10–25 shows how you would use a FOR loop so that all product values currently in status are copied to the relational table named products. Example 10–25 will run much more efficiently when the INSERT statement is compiled with the PREPARE statement. Example 10–26 shows the PREPARE statement being used to compile the INSERT statement with a name of write_products, which is then run by an EXECUTE statement within the FOR loop. Example 10–27 shows the PREPARE statement being used to compile the INSERT statement for direct insert (DIRECT=YES).

Example 10–25 Inefficient FOR Loop

```
FOR prod
   DO
      SQL INSERT INTO products -
         VALUES(:aw_prod_id, :aw_product_name)
      IF SQLCODE NE 0
         THEN BREAK
   DOEND
```

Example 10–26 Improving Efficiency Using Precompiled Code

```
SQL PREPARE write_products FROM -
   INSERT INTO products -
      VALUES(:aw_prod_id, :aw_product_name)
      .
      .
      .
FOR prod
   DO
      SQL EXECUTE write_products
      IF SQLCODE NE 0
         THEN BREAK
   DOEND
```

Example 10–27 Improving Efficiency Using a Direct Insert

```
SQL PREPARE write_products FROM -
   INSERT INTO products -
      VALUES(:aw_prod_id, :aw_product_name)
      DIRECT=YES
      .
      .
      .
FOR prod
   DO
      SQL EXECUTE write_products
      IF SQLCODE NE 0
         THEN BREAK
   DOEND
```

Conditionally Updating a Relational Table

You can also use the values of an analytic workspace variable to update the values in a relational table. Using a `FOR` loop, your OLAP DML program steps through the specified dimension value by value and uses a `WHERE` clause to point to the corresponding row in the relational table.

The program fragment in Example 10–28 updates only those rows in the `products` table where the values in the `prod_id` column match the `aw_prod_id` dimension values currently in status.

Example 10–28 Conditionally Updating a Relational Table

```
FOR prod
   DO
      SQL UPDATE products -
      SET product_name = :aw_newproduct_name -
         WHERE prod_id = :aw_prod_id
      IF SQLCODE NE 0
         THEN BREAK
   DOEND
```

Using Stored Procedures and Triggers

Support is provided for stored procedures and triggers. They cannot contain `SELECT` statements. an analytic workspace stored procedure cannot contain output variables or transactions, nor can it call another procedure. You can create a stored procedure or trigger in an OLAP DML program. Example 10–29 shows the OLAP DML syntax for creating a procedure named `new_products`.

OLAP DML syntax differs slightly from the standard SQL syntax. A tilde (~) is required instead of a semicolon as a terminator, and two colons (`::`) are required instead of one in an assignment statement.

Example 10–29 Creating a Stored Procedure Named new_products

```
SQL CREATE PROCEDURE new_products -
   (aw_id CHAR, aw_name CHAR, aw_cost NUMBER) IS -
      price number~ -
   BEGIN -
      aw_price ::= aw_cost * 2.5~ -
      INSERT INTO products -
         VALUES(aw_id, aw_name, aw_price)~ -
   END~
```

Executing a stored procedure

You use a `PROCEDURE` statement to run a stored procedure, using the following syntax.

```
SQL PROCEDURE procedure-name (arg1, arg2, arg3, . . .)
```

The arguments can be literal text or input host variables. When you use input host variables, be sure to use a colon before the variable name. Also be sure to use the same number of arguments with appropriate data types for the parameters defined in the procedure. You can use literal arguments when executing a stored procedure as shown in Example 10–30 which uses the `new_products` procedure to insert a single row in the `products` table, or you can specify analytic workspace objects as arguments as shown in Example 10–31 which runs the same procedure but inserts data stored in analytic workspace dimensions and variables into the `products` table. The `add-prods` program in Example 10–31, "Using Workspace Objects as Parameters for a Stored Procedure" illustrates using a `FOR` loop to loop over all of the values in status. To call `add_prods`, you issue a command like the following to set the status of `prod` to include only the values you wish to update.

```
CALL add_prods('last 5')
```

Example 10–30 Providing Literal Values to a Stored Procedure

```
SQL PROCEDURE new_products -
   ('P81', '8mm Camcorder')
```

Example 10–31 Using Workspace Objects as Parameters for a Stored Procedure

```
DEFINE add_prods PROGRAM
LD Add new products using stored procedure new_products
PROGRAM
ARG newprods TEXT
PUSH aw_prod
LIMIT aw_prod TO &newprods

" Loop over aw_prod to insert the data
FOR aw_prod
   DO
      SQL PROCEDURE new_products(:aw_prod_id, :paw_rod_name)
   DOEND
POP aw_prod
END
```

Checking for Errors

Although the OLAP DML will signal some SQL errors, it does not automatically signal an error when there is an error in a SQL statement. Instead, the OLAP DML provides support to help you handle errors that are returned.

In your programs, you will need to provide the logic for handling SQL errors. The OLAP DML provides two options, `SQLCODE` and `SQLERRM`, whose values reflect the `SQLCODE` and `SQLERRM` values set in the database.

SQLCODE Option

`SQLCODE` contains an integer error code number. Your programs should test the value of `SQLCODE` after every SQL command to make sure that the command executed successfully. You can also test the value of `SQLCODE` to determine whether you need to break out of a loop. `SQLCODE` typically has one of the values shown in Table 10–3.

Table 10–3 Values of SQLCODE

Code	Meaning
0 (zero)	The last SQL operation was successful.
100	All requested rows have been fetched.
-1	An error has occurred.
Any value that is not 0 or not 100	An error has occurred.

SQLERRM Option

The `SQLERRM` option contains the error message associated with the current error code. It identifies the condition that caused an error to occur. You can control whether or not this message is sent automatically to the current outfile. When you are debugging a program, you will probably want all SQL error messages sent to the current outfile so that you can see them immediately. However, when your application is in use, you will want to suppress the error messages and handle the error condition in a way more suited to your application.

SQLMESSAGES Option

The SQLMESSAGES option controls whether SQL messages are sent to the current outfile, which is usually the screen. To send SQL messages to the current outfile, issue the following command.

```
SQLMESSAGES = yes
```

11

Reading Data from Files

This chapter describes how to read data from external files. It includes the following topics:

- Introducing Data-Reading Programs
- Reading Files
- Specifying File Names in the OLAP DML
- Reading Data from Files
- Reading and Maintaining Dimension Values
- Processing Input Data
- Processing Records Individually
- Processing Several Values for One Variable

Introducing Data-Reading Programs

There is a group of commands, often referred to as data-reading commands, that you can use in programs to read data from external files in various formats: binary, packed decimal, or text.

While some of the data-reading commands can be used individually, it is best to place them in a program that is often referred to as a data-reading program. In this way you can minimize mistakes in typing and test your commands on smaller sets of data. A program also allows you to perform operations in which several commands are used together to loop over many records in a file.

The data-reading commands are described below.

Function or Command	Description
FILEERROR function	Returns information about the first error that occurred when you are processing a record from an input file with the data-reading commands FILEREAD and FILEVIEW.
FILENEXT function	Makes a record available for processing by the FILEVIEW command. It returns YES when it is able to read a record and NO when it reaches the end of the file.
FILEREAD command	Reads records from an input file, processes the data, and stores the data in workspace dimensions, composites, relations, and variables, according to descriptions of the fields in the input record.
FILEVIEW command	Works in conjunction with the FILENEXT function to read one record at a time of an input file, process the data, and store the data in workspace dimensions and variables according to the descriptions of the fields.
RECNO function	Reports the current record number of a file opened for reading.

You use the data-reading commands with file I/O commands, such as the commands described below.

Function or Command	Description
FILECLOSE command	Closes an open file.
FILEGET function	Returns text from a file that has been opened for reading.
FILEOPEN function	Opens a file, assigns it a fileunit number (an arbitrary integer), and returns that number.
FILEPUT command	Writes data that is specified in a text expression to a file that is opened in WRITE or APPEND mode.
FILEQUERY function	Returns information about one or more files.
FILESET command	Sets the paging attributes of a specified fileunit.

Reading Files

While reading from a file, you can format the data from each field individually, and use DML functions to process the information before assigning it to a workspace object. Reading a file generally involves the following steps.

1. Open the file you want to read.
2. Read data from the file one record or line at a time.
3. Process the data and assign it to one or more workspace objects.
4. Close the file.

The FILEREAD and FILEVIEW commands have the same attributes and can do the same processing on your data. However, they differ in important ways:

- The FILEREAD command loops automatically over all records in the file and processes them automatically. Use FILEREAD when all records in the file are the same.
- The FILEVIEW command processes one record at a time. Use FILEVIEW when there is more than one type of record in the file.

Creating a Program to Read Data

The following table shows, for each method, the commands you need to open and close the input file, to read the file, and to handle errors that might occur.

Program Section	FILEREAD	FILEVIEW
Initialization	`VARIABLE funit INTEGER` `TRAP ON error`	`VARIABLE funit INTEGER` `TRAP ON error`

Body	`funit = FILEOPEN(-`
	`   'alias/datafile' READ)`
	`FILEREAD funit`
	`   .`
	`   .`
	`   .`
	`FILECLOSE funit`
		`funit = FILEOPEN(-`
		`   'alias/datafile' READ)`
		`WHILE FILENEXT(funit)`
		`   DO`
		`   FILEVIEW funit . . .`
		`   DOEND`
		`FILECLOSE funit`

Normal Exit	`RETURN`	`RETURN`

Abnormal Exit	`error:`
	`IF funit NE na`
	`   THEN FILECLOSE funit`
		`error:`
		`IF funit NE na`
		`   THEN FILECLOSE funit`

Note: The error handling in the abnormal exit section of the programs closes the file only when the file is open. The FILEOPEN function signals an error when for any reason the system cannot open the file. The program tries to close the file after the ERROR label only when FUNIT holds a valid file unit number. You can add additional commands to the error handling section as well. These sections of the program are the same for both methods.

Specifying File Names in the OLAP DML

The FILEOPEN function opens a file and returns an integer that uniquely identifies that file. This file identifier is known as a **fileunit**. Once you have opened a file and obtained a fileunit, all subsequent calls to data-reading commands and file I/O commands for that file reference the fileunit instead of the file name.

A file identifier is a character string that specifies a file stored on disk. The file identifier includes the directory alias and the file name; these two components are separated by a forward slash (/). You can use the CDA command to specify a current

directory alias. In this case, you do not have to specify a directory alias in a file identifier, because Oracle OLAP assumes that the file is in your current directory alias. Contact your Oracle DBA for access rights to a directory alias where you can read and write files.

When specifying file identifiers in OLAP DML commands, it is good practice to always enclose them in single quotation marks. This will prevent parsing errors in cases where file name components are also workspace object names or reserved words.

Reading Data from Files

Data-reading programs read data from a file, record-by-record, and assign that data to variables, relations, dimensions, and composites in your analytic workspace. When the records in the file contain dimension values, you can limit dimensions to these values with the `FILEREAD` command before assigning the data to a variable dimensioned by them.

Example 11–1 Using FILEREAD in a Data-Reading Program

Suppose you want to update unit sales data for the `product` dimension in an analytic workspace. The new sales information is stored in a file called `units.dat`, which has the layout shown in the following figure.

```
                  1 1 1 1 1 1 1 1 1 1 2 2 2
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
DISTRICT        PRODUCT         Unit Sales
```

Columns	Description
1 - 8	District Names
9 - 16	Product Names
17 - 22	Unit Sales data

The `FILEREAD` command that reads the sample `units.dat` file is shown below.

```
FILEREAD funit -
    COLUMN 1 WIDTH 8 district -
    COLUMN 9 WIDTH 8 product -
    COLUMN 17 WIDTH 6 units
```

This command is processed in these steps:

1. The field is read beginning in column 1, and `district` dimension is limited to the value read. When the value read is not a dimension value of `district`, an error occurs.
2. The second field is read, and the `product` dimension is limited.
3. The third field is read, and the value is assigned to the `units` variable in the cell corresponding to the district and product read in Steps 1 and 2.

The full program, with commands to open and close the file, is shown next.

```
DEFINE readit1 PROGRAM
LD Read a data file
VARIABLE funit INTEGER
TRAP ON error
funit = FILEOPEN('olapfiles/units.dat' READ)
FILEREAD funit -
   COLUMN 1 WIDTH 8 district -
   COLUMN 9 WIDTH 8 product -
   COLUMN 17 WIDTH 6 units
FILECLOSE funit
RETURN
error:
IF funit NE na
   THEN FILECLOSE funit
END
```

Reading Structured PRN Files

You can also use the data-reading commands to read structured PRN files, which are produced by many PC software products. In a PRN file, quoted text or a series of numbers demarcated by spaces or commas constitutes a field of the record. Instead of specifying the column in which a field starts, you can use the `STRUCTURED` keyword to specify that you are reading a structured file. You can also use one or more `FIELD` keywords to indicate the number of the field you want to read.

Example 11–2 Reading a Structured PRN File

Suppose you want to read sales data from the structured PRN file illustrated below.

```
010195  "TENTS"   "BOSTON"    307  50808.96
010195  "TENTS"   "ATLANTA"   279  46174.92
010195  "TENTS"   "CHICAGO"   189  31279.78
```

```
010195  "TENTS"   "DALLAS"    308  50974.46
010195  "TENTS"   "DENVER"    215  35582.82
010195  "TENTS"   "SEATTLE"   276  45678.41
010195  "CANOES"  "BOSTON"    352  70489.44
010195  "CANOES"  "ATLANTA"   281  56271.40
010195  "CANOES"  "CHICAGO"   243  48661.74
010195  "CANOES"  "DALLAS"    176  35244.72
010195  "CANOES"  "DENVER"    222  44456.41
010195  "CANOES"  "SEATTLE"   335  67085.12
```

The file has `product` values in the second field, `district` values in the third field, and sales data in the fifth field.

You can limit the `month` dimension to the desired month, and then use the following command to read the sales data from the first six records in the file.

```
FILEREAD unit STOPAFTER 6 STRUCTURED FIELD 2 product -
   district FIELD 5 sales
```

Reading and Maintaining Dimension Values

The records in a data file often contain dimension values, which are used to identify the cell in which the data values should be stored. When all of the dimension values in the file already exist in your analytic workspace, you can use the default attribute `MATCH` in the dimension field description. `MATCH` accepts only dimension values that already are in the analytic workspace.

When `FILEREAD` finds an unrecognized value, the command signals an error that warns you about the bad data. Your data-reading program can handle the error by skipping the data and continuing processing, or by halting the processing and letting you check the validity of the data file.

Example 11–3 Reading Records Only for Existing Dimension Values

The following example shows a data file that contains 6-character values for the dimension `productid`, names for each product, and the number of units sold.

```
1234AA00CHOCOLATE CHIP COOKIES        123
1099BB00OATMEAL COOKIES               145
2344CC00SUGAR COOKIES                 223
3222DD00BROWNIES                      432
5553EE00GINGER SNAP COOKIES           233
```

The following workspace objects are used by the example program.

```
DEFINE productid DIMENSION ID
DEFINE productname VARIABLE TEXT <productid>
DEFINE units.sold VARIABLE INTEGER <month productid>
```

The `dr.prog` program reads the file. The values of `productid` with the associated product name are already part of the analytic workspace, so the program uses the `productid` values only to set status and assign the units data to the right cells of the `units.sold` variable.

The `MATCH` attribute is left out of the field description because it is the default. When the program finds a value for `productid` that is not in the analytic workspace, it branches to the trap label. If the user interrupts the program (that is, the error name is `attn`) or the data file cannot be opened, then the program ends. Otherwise, the program resets the error trap and branches back to `FILEREAD` to continue with the next record.

The example program, named `dr.prog`, has the following definition.

```
DEFINE dr.prog PROGRAM
LD Reads a file with existing dimension values
PROGRAM
VARIABLE funit INTEGER
TRAP ON error
PUSHLEVEL 'save'
PUSH month productid
LIMIT month TO FIRST 1
funit = FILEOPEN('olapfiles/dr.dat' READ)
next:
FILEREAD funit -
   COLUMN 1 WIDTH 6 productid -
   COLUMN 39 WIDTH 3 units.sold
FILECLOSE funit
POPLEVEL 'save'
RETURN
error:
"Skip current record and continue processing
IF funit NE na and ERRORNAME NE 'ATTN'
   THEN DO
     TRAP ON error
     GOTO next
     DOEND
```

```
"Close the file
IF funit NE na
   THEN FILECLOSE funit
POPLEVEL 'save'
END
```

Adding New Dimension Values from a Data File

When your data file contains a mixture of existing and new dimension values, you can add the new values and all the associated data to the analytic workspace by using the APPEND attribute in the field description.

Example 11–4 Adding New Dimension Values from a Data File

The first FILEREAD command in the dr.prog2 program uses APPEND to add any new productid values to the analytic workspace. The second FILEREAD command includes a field to read the product name so the new data will be complete.

The dimension maintenance performed by APPEND might be done in the same FILEREAD command that reads the data, but that would cause inefficient handling of the data. The data is handled more efficiently when the dimension maintenance and data reading are performed in two separate passes over the file.

The error processing in this version is shorter because there is no need to skip nonexistent product values and branch back. If there is an error, then the program closes the file, restores any pushed values, and terminates.

The program, named dr.prog2, has the following definition.

```
DEFINE dr.prog2 PROGRAM
LD Reads a file with new dimension values
PROGRAM
VARIABLE funit INTEGER
TRAP ON error
PUSHLEVEL 'save'
PUSH month productid
LIMIT month TO FIRST 1
funit = FILEOPEN('olapfiles/dr.dat' READ)
FILEREAD funit COLUMN 1 APPEND WIDTH 6 productid
FILECLOSE funit
funit = FILEOPEN('olapfiles/dr.dat' READ)
FILEREAD funit -
   COLUMN 1 WIDTH 6 productid -
   COLUMN 9 WIDTH 30 productname -
   COLUMN 39 WIDTH 3 units.sold
```

```
FILECLOSE funit
POPLEVEL 'save'
RETURN
error:
IF funit NE na
   THEN FILECLOSE funit
POPLEVEL 'save'
END
```

Reading Dimension Values by Position

If the target dimension has a data type of TEXT, NTEXT, or ID and the input field in the file contains dimension position numbers (rather than dimension values), then you must specify a conversion type of INTEGER in the field description. The conversion type specifies how input data should be converted to values of the target dimension.

Suppose the target dimension is month, then you can use the following command to read input values that represent positions within the default status of month.

```
FILEREAD unit COLUMN 1 WIDTH 8 INTEGER month
```

When the input field contains position numbers, you cannot use the APPEND keyword to add new values to a target dimension.

The Use of Composites

Composites are automatically maintained. The way in which you define and use composites can dramatically improve or hinder performance. The more you know about analytic workspace design, especially in regard to the applications that will be used with an analytic workspace, the more effective your use of composites will be.

Reading and Maintaining Conjoint Dimensions

When you have conjoint dimensions in your analytic workspace, you can set the status of those dimensions while reading a file with the FILEREAD command. Typically, the records in the data file will have a separate field for each base dimension of your conjoint dimension. For example, a file might have a market name in the first field, a product name in the second, and then one or more fields containing sales data.

Example 11–5 Reading and Maintaining Conjoint Dimensions

To read the sales data into a variable dimensioned by a conjoint dimension, for example `markprod`, you can use a `FILEREAD` command as follows.

```
FILEREAD funit markprod -
   = <W 8 market W 8 product> W 10 sales
```

This command will read a value of the `market` dimension from the first 8-character field of the record and a value of the `product` dimension from the next 8-character field.

The command will then use the results to set the status of `markprod`, which is a conjoint dimension defined as follows.

```
DEFINE markprod DIMENSION <market product>
```

The command then reads the last field and assigns the value to the variable `sales`, which is dimensioned by `markprod`.

By including the `APPEND` keyword in the field description, you can add new values to `market`, `product`, and `markprod`, when the `FILEREAD` command encounters values in the file that do not match existing dimension values.

```
FILEREAD funit APPEND markprod -
   = <W 8 APPEND market W 8 APPEND product> W 10 sales
```

Translating Coded Dimension Values

The fields containing dimension information in your data file might have values that are not identical to the dimension values in your analytic workspace. The file values might be abbreviated or otherwise encoded. The way you translate a coded dimension value varies depending on whether the code is merely an abbreviation (for example, “P” for `product`) or if the code is more complicated.

When the file contains an abbreviated code, you can sometimes complete the value by using the `RSET` or `LSET` attribute to add text to the right or left of the value in the file.

For example, products in the file might be identified by all-numeric product numbers, while in your analytic workspace, the values of the `product` dimension might be these same product numbers preceded by the letter `P`. In this case, you can use the `LSET` attribute to add the letter `P` to the values in the file.

```
FILEREAD funit COLUMN 1 WIDTH 6 LSET 'P' product
```

The letter `P` is added when the value is read from the file; it is not added when the modified value is matched with or assigned to the `product` dimension.

To correctly read values that have less straightforward codes, you can set up another dimension containing the coded values found in the data file, along with a relation to the real dimension. `FILEREAD` can then use the relation to determine the actual dimension value. Or you can use any OLAP DML function to alter or manipulate the coded value to make it match a value in your analytic workspace.

When reading coded data that must be manipulated in some way before being stored in the target, use an assignment statement (shown below) in the field description.

```
target = expression
```

The *expression* argument specifies the processing or calculation to be performed. If you want to include the value just read from the file as part of the *expression*, then use the `VALUE` keyword.

Both of the following field descriptions function identically.

```
COLUMN n WIDTH n target

target = COLUMN n WIDTH n VALUE
```

Example 11–6 Translating Codes into Dimension Values

This example illustrates the use of an expression for translating codes into dimension values.

The following example shows the data file, which has 3-character codes for months, and 2-character codes for districts and products.

```
SEP BO CH 113945 115
OCT BO CH 118934 115
SEP BO CO  92013 119
OCT BO CO  95820 119
SEP BO WI  83201 110
OCT BO WI  82986 110
SEP DA CH 111792 115
OCT DA CH 136031 114
SEP DA CO  91641 121
OCT DA CO  96347 120
SEP DA WI  89734 109
OCT DA WI  88264 109
```

The following OLAP DML objects are used by the example program.

```
DEFINE distcode DIMENSION ID
DEFINE district.dcode RELATION district <distcode>
DEFINE prodcode DIMENSION ID
DEFINE Product.pcode RELATION product <prodcode>
```

The example program, named `dr.prog3`, has the following definition.

```
DEFINE dr.prog3 PROGRAM
LD Translates coded values into valid dimension values
PROGRAM
VARIABLE funit INT
funit = FILEOPEN('olapfiles/dr3.dat' READ)
FILEREAD funit -
   COLUMN 1 WIDTH 3 APPEND RSET '96' month
FILECLOSE funit
funit = FILEOPEN('olapfiles/dr3.dat' READ)
FILEREAD funit -
   COLUMN 1 WIDTH 3 RSET '96' month -
   COLUMN 5 WIDTH 2 district = district.dcode -
     (distcode VALUE) -
   COLUMN 8 WIDTH 2 product = product.pcode -
     (prodcode VALUE) -
   COLUMN 11 WIDTH 6 STRIP units -
   COLUMN 18 WIDTH 3 SCALE 2 price
FILECLOSE funit
END
```

The program translates the 2-character codes for districts and products into values of a `district` dimension and a `product` dimension. The program also appends a 2-digit year to the months.

In the first `FILEREAD` command, the `APPEND` keyword is used so that new months are added to the MONTH dimension.

```
FILEREAD fileunit COLUMN 1 WIDTH 3 APPEND RSET '96' month
```

For the district and product fields, the program reads the value from the data file and finds the corresponding dimension value using the relations `district.dcode` and `product.pcode`.

```
COLUMN 5 WIDTH 2 district = district.dcode distcode VALUE)
COLUMN 8 WIDTH 2 product = product.pcode (prodcode VALUE)
```

The program uses a QDR with the keyword VALUE representing the code read from the data file. For the districts, the distcode VALUE QDR modifies the relation district.dcode, which holds district names. It specifies the district that corresponds to the value of distcode just read from the data file. The QDR for product works the same way.

The program assumes the product and district dimension values are already in the analytic workspace, along with the distcode and prodcode dimensions and the relations connecting them to district and product. Once the coded values have been processed, the resulting values of district and product are used to limit the dimension status so that the data is put in the right cells of the units and price variables.

Finally, you can see in the data file that the price data, which starts in column 18, does not have a decimal point. The SCALE attribute on the last line of the FILEREAD command puts two decimal places in each price value.

Processing Input Data

Assignment statements created with the = command have a wide application in the data-reading commands. With the = command you can process any value read from a file in a variety of ways. Instead of just assigning the values as read to a variable or relation, you can modify those values to make them more suitable to your application.

The expression you use can be as simple or complex as you need. You can even perform conditional processing on the values read, based on other data already stored in your analytic workspace or previously read from the file.

For an example of using FILEREAD commands using an assignment statement in a field description, see "Reading and Maintaining Dimension Values" on page 11-7.

Example 11–7 Modifying Values Read from a File

The following command reads sales data and assigns it to the variable sales, replacing whatever value is already stored in that variable.

```
FILEREAD funit W 8 district W 8 product W 10 sales
```

Using an expression, however, you can add the new data to the value currently stored in the variable.

```
FILEREAD funit W 8 district W 8 product sales -
  = sales + W 10 VALUE
```

The data just read from the file is represented in the expression by the keyword VALUE.

Suppose you have two different types of records in a file, you can read different fields for each type of record.

```
FILEREAD funit W 1 rectype W 8 district W 8 -
   APPEND product -
   prodname = -
     IF rectype EQ 'A' THEN COL 25 W 16 VALUE -
     ELSE COL 42 W 16 VALUE
```

Specifying a Conversion Type for Data

In general, you do not need to specify a data type when you read input values into a workspace variable. By default, input values are converted to the data type of the target variable.

However, when the target variable has a data type of DATE, you can use either the default conversion type of DATE or an alternative conversion type of RAW DATE.

You might also want to specify a conversion type when you use an expression to process input values before storing them in a target variable.

Processing Records Individually

Your data files do not always have the same type of data in every record. You might find that you need different field descriptions and different target objects for each record, or you might have two or more distinct types of records mixed together in a single file. You might even have to decide what to do with the data in a record based on the contents of one or more of its fields.

The FILENEXT function and the FILEVIEW command allow you to retrieve one record at a time from a file and look at its data one or more times. FILENEXT is a Boolean function, which reads a record from the data file. It returns YES when it finds a record and NO when it reaches the end of the file. The record read by FILENEXT is then available to process with the FILEVIEW command.

Typically, FILENEXT is used as the condition of a WHILE command, so that the data-reading program continues reading until it reaches the end of the file and finds no more records. Within the WHILE loop, the FILEVIEW command is used one or more times to process data from any field in the current record. Often the operation of a FILEVIEW command depends on the data processed by a previous command in the WHILE loop.

Example 11–8 Reading Different Data from the Same Record

In the data shown in the following example, the second field of each record contains the name of the target variable for the data in the last field.

```
CEREALS     DOL         VS100       US    JUN96    5000000
CEREALS     LBS         VS100       US    JUN96    4800000
CEREALS     CASE        VS100       US    JUN96     180000
CEREALS     DOL         VS100       BOS   JUN96      62500
CEREALS     LBS         VS100       BOS   JUN96      62830
CEREALS     CASES       VS100       BOS   JUN96       2750
CEREALS     DOL         VS100       CHI   JUN96      75290
CEREALS     LBS         VS100       CHI   JUN96      73000
CEREALS     CASES       VS100       CHI   JUN96       2700
CEREALS     DOL         VS100       LASF  JUN96     143070
CEREALS     LBS         VS100       LASF  JUN96     150500
CEREALS     CASES       VS100       LASF  JUN96         NA
```

The following OLAP DML objects are used by the example program.

```
DEFINE dol VARIABLE DECIMAL <month item market>
DEFINE lbs VARIABLE INTEGER <month item market>
DEFINE cases VARIABLE INTEGER <month item market>
```

The `dr.prog4` program tests records against criterion before getting values. In the program, the first `FILEVIEW` command gets the name of the variable and stores it in a local variable named `varname`. The second `FILEVIEW` command gets the value and assigns it to the object specified in `varname`.

The example program, named `dr.prog4`, contains the following code.

```
VARIABLE funit INTEGER
VARIABLE varname TEXT
funit = FILEOPEN('olapfiles/dr4.dat' READ)
WHILE FILENEXT(funit)
  DO
    FILEVIEW funit COLUMN 13 WIDTH 12 varname
    FILEVIEW funit COLUMN 25 WIDTH 12 item -
                   COLUMN 37 WIDTH  6 market -
                   COLUMN 43 WIDTH  5 month -
                   COLUMN 48 WIDTH 10 &varname
  DOEND
FILECLOSE funit
```

Reading Different Records

You might want to process only some of the records in a file, based on some criterion in the record itself. You can use one FILEVIEW command to check a field for an appropriate value and, if it is found, then you can process the rest of the record with a second FILEVIEW command.

When the record does not meet the criterion for processing, you can save it in another file using the FILEPUT command. FILEPUT with the FROM keyword writes the last record read by FILENEXT directly to the designated output file. You can also use a FILEPUT command in the error section of your program to keep track of any records that could not be processed because of errors.

Before you use FILEPUT in your data-reading program, you must open a second file in write mode. At the end of the program, you must close it.

Processing Several Values for One Variable

Sometimes several contiguous fields in a file contain data values that you want to assign to the same variable. Each field corresponds to a different value of one of the dimensions of the target variable.

For repeating fields, you can use an ACROSS phrase in the field description to read the successive fields and place the values in the appropriate cells of the target variable. The ACROSS phrase extracts data for each dimension value in the current status or until it reaches the end of the record. You can limit the ACROSS dimension before the FILEREAD (or FILEVIEW) command, or you can limit it temporarily in the ACROSS phrase.

When the data file contains the information you need to limit the ACROSS dimension, you can extract the dimension values using a temporary variable, limit the dimension, and then read the rest of the file.

Example 11–9 Assigning Multiple Fields to the Same Variable

Successive fields might hold sales data for successive months, as shown in the layout of `unitsale.dat` in the following figure.

```
                 1 1 1 1 1 1 1 1 1 1 1 2 2 2 . . . 7 7 7 7 7 7 8
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 . . . 4 5 6 7 8 9 0
PRODUCT    | JAN96 | FEB96 | . . . | DEC96
           |        Unit Sales Data
```

Columns	Description
1 - 8	Product Names
9 - 14	Unit sales for January 1996
15 - 20	Unit sales for February 1996
.	.
.	.
.	.
75 - 80	Unit sales for December 1996

In the `unitsale.dat` file, columns 9 through 80 contain twelve 6-character fields. Each field contains sales data for one month of 1996.

The full data-reading program, with commands to open and close the file, is shown next.

```
DEFINE dr.prog5 PROGRAM
LD Read a data file
VARIABLE funit INTEGER
TRAP ON error
funit = FILEOPEN('olapfiles/unitsale.dat' READ)
FILEREAD funit -
   COLUMN 1 WIDTH 8 product -
      ACROSS month jan96 TO dec96: WIDTH 6 units
FILECLOSE funit
RETURN
error:
IF funit NE na
   THEN FILECLOSE funit
END
```

The `ACROSS` phrase reads each of these fields into separate cells in the `units` variable.

```
ACROSS month jan96 TO dec96: WIDTH 6 units
```

The `FILEREAD` command reads the sample `unitsale.dat` file.

```
FILEREAD funit -
   COLUMN 1 WIDTH 8 product -
      ACROSS month jan96 TO dec96: WIDTH 6 units
```

This command first reads the field beginning in column 1 and limits the `product` dimension to the value read. (When the value read is not a dimension value of `product`, an error occurs.) The command then reads the next 12 fields and assigns the values read to the `units` variable for each month of 1996.

Example 11–10 Using Input Data to Limit the ACROSS Dimension

As shown in following example, the first record of the data file contains values of `month` as labels for each column of data.

```
                   JAN96      FEB96      MAR96      APR96
TENT            50,808.96  34,641.59  45,742.21  61,436.19
CANOES          70,489.44  82,237.68  97,622.28 134,265.60
RACQUETS        56,337.84  60,421.50  62,921.70  74,005.92
SPORTSWEAR      57,079.10  63,121.50  67,005.90  72,077.20
FOOTWEAR        95,986.32 101,115.36 103,679.88 115,220.22
```

The following workspace objects are used by the example program.

```
DEFINE enum DIMENSION INTEGER
DEFINE monthname VARIABLE ID <enum> TEMPORARY
DEFINE salesdata VARIABLE DECIMAL <month product>
```

The example program, named `dr.prog6`, has the following definition.

```
DEFINE dr.prog6 PROGRAM
PROGRAM
VARIABLE funit INTEGER
TRAP ON cleanup
PUSHLEVEL 'save'
PUSH month product
funit = FILEOPEN('olapfiles/dr6.dat' READ)
IF FILENEXT(funit)
   THEN FILEVIEW funit COLUMN 16 ACROSS enum: -
     W 11 monthname
LIMIT month TO CHARLIST(monthname)
FILEREAD funit W 15 product COLUMN 16 ACROSS month: -
   W 11 salesdata
```

```
cleanup:
FILECLOSE funit
POPLEVEL 'save'
END
```

The program does not know how many months the file contains. The program uses a temporary variable dimensioned by an INTEGER dimension to read the month names from the file. The INTEGER dimension enum must have at least as many values as the largest data file has months.

FILENEXT reads only the first record. The CHARLIST function creates a list of the month names, which is used to limit the month dimension.

Finally, the FILEREAD command processes the rest of the record using month as the ACROSS dimension. All the sales data is assigned to the correct months without the user having to specify them.

12

Aggregating Data

This chapter describes how to use the aggregation features of the OLAP DML. This chapter includes the following topics:

- About Aggregating Detail Data
- Preliminary Steps Prior to Aggregation
- Creating an Aggregation Map
- About the RELATION Command
- Aggregating Non-Hierarchical Data
- How to Generate Precalculated Data
- How to Calculate Data at Runtime
- Creating Custom Aggregates
- Balancing Precalculated and Runtime Aggregation
- Performing Partial Aggregations
- Combining AGGREGATE with Forecasts and Programs

About Aggregating Detail Data

Business analysis applications typically use hierarchical dimensions for their data. In Oracle OLAP, all members of a hierarchy, regardless of their level, are stored in a single dimension. A self-relation and a parent relation identify the parent-child relationships among the members. Other, nonhierarchical dimension (such as a line item dimension) may require a model to calculate the values.

Data at the detail level is typically acquired from another source (such as a transactional database or flat files), but the aggregate data must be calculated. These calculations can be done in two distinct ways:

- As a data maintenance procedure. The DBA acquires detail data, calculates the aggregate values, and stores them in the analytic workspace for all users to share. This type of aggregate data is sometimes call **precomputed** or **stored** aggregates. It supports the fastest querying time, but increases the size of the analytic workspace and therefore the size of the relational database. The amount of precomputed data may also be limited by the amount of time available for the data task (often called a **batch window**).
- At run-time when needed. The cells for the aggregate values are NA (that is, they are empty) until a query requests the aggregate values. The aggregates are then computed in response to the query. The results can be stored in a temporary cache for use throughout the session. If the session has write access to the analytic worksheet, the results can also be stored permanently. This type of aggregate data is referred to as **on-the-fly** or **run-time** aggregates. It slows querying time since the data must be calculated instead of just retrieved, but it does not require permanent storage for aggregate values.

Oracle OLAP supports both types of aggregation, and provides a mechanism for precomputing some values and calculating others at run-time within a single data variable.

> **See Also:** "Defining Hierarchical Dimensions and Variables That Use Them" on page 3-22 for more information about hierarchical dimensions.

Functionality Available with AGGREGATE

The OLAP DML supports a variety of aggregation methods including first, last, average, weighted average, and sum. In a multidimensional variable, the aggregation method can vary by dimension.

When variables are dimensioned with detailed, multilevel hierarchies, the number of cells of aggregate data can be many times greater than the number of cells of detail data. In contrast, users typically query some levels of data heavily and other levels very infrequently. They tend to focus on top-level aggregates and only occasionally drill to middle-level aggregates, although the middle-level aggregates comprise the largest proportion of aggregate data.

For this reason, the OLAP DML provides an aggregation method that allows some of the data to be aggregated and stored, while other data is aggregated at runtime. The DBA can choose whatever method seems appropriate: by level, individual member, member attribute, time range, data value, or other criteria. A technique called "skip level" aggregation pre-aggregates every other level in a dimension hierarchy. It is described in "Calculating Data Using the Skip-Level Approach" on page 12-25.

Table 12–1 lists commands that support aggregation in the OLAP DML.

Table 12–1 Commands That Support Aggregation

Command	Description
AGGREGATE command	Calculates data for permanent storage in the analytic workspace.
AGGREGATE function	Calculates data on-the-fly in response to a query.
AGGMAP command	Adds contents to an aggmap object that identify which aggregates are calculated by the AGGREGATE command and which ones are calculated by the AGGREGATE function. It also identifies whether the run-time aggregates are cached for use throughout the session. This decision has implications for whether run-time changes to the detail values are reflected in the aggregate values.
AGGMAPINFO command	Returns information about the contents of an aggregation map object, such as whether it contains commands for aggregation or allocation.
CLEAR command	Clears data values in the aggregate cache.
MULTIPATHHIER option	Controls whether detail data can be aggregated over multiple paths.
POUTFILEUNIT option	Identifies a location that receives information on the progress of an AGGREGATE command.
SESSCACHE option	Controls whether an aggregate cache persists throughout a session.
VARCACHE option	Controls how on-the-fly aggregates are stored.

Process Overview: Aggregation

These are the basic steps you need to follow to generate and manage aggregate data:

1. Perform the initial analysis of your data, as described in "Preliminary Steps Prior to Aggregation" on page 12-4, to make sure that it is set up properly.
2. Create an aggregation map that identifies which data will be precalculated and which data will be calculated as needed. Identify variables that are dimensioned identically, because they can share an aggregation map.
3. Set the `POUTFILEUNIT` option so that you can monitor the progress of the aggregation.
4. Use the `AGGREGATE` command with the aggregation map to precalculate the data and store it in the database.
5. If the aggregation map specifies run-time calculations, then:
   a. Compile the aggregation map.
   b. Add a property to the variable that will trigger the `AGGREGATE` function in response to a runtime request for data.

These steps are described in detail in this chapter.

Preliminary Steps Prior to Aggregation

There are several pre-aggregation steps that you should perform to achieve the best performance:

- Get the names of self-relations or the names of parent and hierarchy relations defined within the DML.
- Check all composite dimensions to make sure that they have BTREE indexes.

Identifying the Parent and Level Relations

All aggregation maps require the identity of the **parent relation** for each dimension that is being aggregated. The parent relation is a self-relation that defines the hierarchy by identifying the parent of each dimension value.

If some of the data will be aggregated at runtime, then you may want to use a **level relation** to distinguish levels that will be omitted from the pre-calculation. The level relation identifies the level of the hierarchy for each dimension value. This relation is needed to identify which levels are precalculated and which ones are calculated

at run-time. Skip-level aggregation is a recommended technique, described in "Balancing Precalculated and Runtime Aggregation" on page 12-24, which uses level relations.

Example 12–1 describes the parent and level relations.

You may be able to use the OBJ function to find out information about a workspace object. For example, the following command may display the name of the level dimension for the geography dimension:

```
REPORT OBJ(PROPERTY 'leveldim' 'geography')
```

Note: This information may or may not be available through the PROPERTY keyword, depending upon the method originally used to create these relations.

If the OBJ function does not yield results, then you must look at the contents of the variables in your analytic worksheet to see if these relations exist, and if not, then create them.

Example 12–1 Identifying the Parent and Level Relations

The following are the object definitions for three dimensions and two relations. These objects provide the information that the aggregation map needs to aggregate data dimensioned by geography.

```
DEFINE GEOGRAPHY DIMENSION TEXT WIDTH 12
LD Geography dimension values

DEFINE GEOGRAPHY.HIERARCHIES DIMENSION TEXT
LD Hierarchy dimension for Geography

DEFINE GEOGRAPHY.LEVELDIM DIMENSION TEXT
LD List of hierarchy levels for GEOGRAPHY

DEFINE GEOGRAPHY.PARENTREL RELATION GEOGRAPHY <GEOGRAPHY GEOGRAPHY.HIERARCHIES>
LD Parent-child relation for Geography

DEFINE GEOGRAPHY.LEVELREL RELATION GEOGRAPHY.LEVELDIM <GEOGRAPHY GEOGRAPHY.HIERARCHIES>
LD Level of each member in each Geography hierarchy
```

The geography dimension contains values at all levels of the hierarchy, such as WORLD, AMERICAS, CANADA, TORONTO, MONTREAL, NEWYORK, CHICAGO, SEATTLE, MEXICO, and so forth.

The geography.hierarchies dimension identifies the names of the hierarchies. For example, geography might have two hierarchies, STANDARD and CONSOLIDATED.

The geography.leveldim dimension identifies the names of the levels, such as CITY, STATE, COUNTRY, REGION, WORLD.

The geography.parentrel relation is a self-relation. For each hierarchy and each dimension value, it identifies the parent value. For example, in the STANDARD hierarchy, the parent of KYOTO is JAPAN, and the parent of JAPAN is ASIA.

The geography.levelrel relation identifies the level for each dimension value in each hierarchy. For example, in the STANDARD hierarchy, KYOTO is at the CITY level, JAPAN is at the COUNTRY level, and ASIA is at the REGION level.

Verifying That All Composites Use BTREE Indexes

You will achieve the best performance results with AGGREGATE when all of the variable's composites use the BTREE index algorithm. You can use the DESCRIBE command to find out if a composite uses BTREE or HASH. If a composite uses HASH, it will be displayed in the composite definition. If a composite uses BTREE, no index algorithm will be displayed in the composite definition, because BTREE is the default algorithm for composites.

The following object definition for the market.prod composite shows that it uses a HASH index:

```
DEFINE MARKET.PROD COMPOSITE <MARKET PRODUCT> HASH
```

To change to a BTREE index, use the CHGDFN command:

```
CHGDFN market.prod BTREE
```

The composite definition looks like this with a BTREE index:

```
DEFINE MARKET.PROD COMPOSITE <MARKET PRODUCT>
```

Creating an Aggregation Map

An aggregation map is a workspace object. You first define the object and then add its contents, similar to creating a model or program. The contents of an aggregation map are commands that specify the data that should be aggregated for each dimension in the variable definition. It also identifies which data should be pre-calculated and which data should be calculated on the fly. Therefore, both the AGGREGATE command and the AGGREGATE function require an aggregation map

To create an aggregation map, you must:

1. Define an aggmap object.
2. Add contents to the aggmap object.

How to Define an Aggmap Object

You can define an aggregation map with the `DEFINE AGGMAP` command. The syntax of the `DEFINE AGGMAP` command is as follows:

```
DEFINE name AGGMAP
```

Where:

name is the name of the aggregation map.

How to Add Contents to an Aggmap Object

After you have defined an aggmap object, you must add contents to it. You can use the following ways to edit an aggregation map. See the examples that follow this list for details.

- Use the `AGGMAP` command to enter or replace the contents of the aggregation map.
- Use the `EDIT AGGMAP` command in OLAP Worksheet.
- Create a text file with the contents of the aggregation map, then use the `INFILE` command to read it into your workspace.

Example 12–2 Using the AGGMAP Command

The following program uses the `JOINLINES` function with the `AGGMAP` command to add `RELATION` commands to an aggmap object.

```
DEFINE AGGTEST PROGRAM
LD Create an aggregation map
PROGRAM
IF NOT EXISTS('test.agg')
     THEN DEFINE test.agg AGGMAP
     ELSE CONSIDER test.agg
```

```
AGGMAP JOINLINES(-
     'RELATION geography.parentrel' -
     'RELATION product.parentrel' -
     'RELATION channel.parentrel' -
     'RELATION time.parentrel' -
     'END')
END
```

Example 12–3 Using the EDIT AGGMAP command in OLAP Worksheet

To use the `EDIT` command in OLAP Worksheet to edit an aggmap object, take these steps:

1. Issue this DML command, where `myaggmap` is the name of an existing aggmap object.

   ```
   EDIT AGGMAP myaggmap
   ```

   The `AGGMAP` edit window will appear.
2. Enter the body of the aggregation map, or make whatever changes you wish to an existing aggregation map.
3. To save your changes, choose **Save** from the File menu.
4. To close the edit window, choose **Quit** from the File menu.

Example 12–4 Using the INFILE Command to Execute Commands in a Text File

You can create a text file that contains the contents of the aggregation map. You can use this text file to create or modify the aggregation map.

Suppose that you have defined an aggmap object named `gpct.aggmap`. You can create a file with these contents:

```
CONSIDER gpct.aggmap
AGGMAP
RELATION geography.parentrel
RELATION product.parentrel
RELATION channel.parentrel
RELATION time.parentrel
END
```

If the file is named `aggmap.inf` in the `userfiles` directory alias, then you can use the following `INFILE` command to execute these commands in your session:

```
INFILE 'userfiles/aggmap.inf'
```

Contents of an Aggregation Map

An aggregation map contains the following commands:

- AGGMAP command indicates the beginning of an aggregation map. Depending upon how you add contents to an aggmap object, you may not need to include this command explicitly.
- RELATION command identifies a parent relation or self-relation (which acts as a hierarchy) of a dimension, which will be used to aggregate data. It can also identify the type of aggregation and the selection of data to be aggregated. By default, all of the data is summed. All aggregation maps contain one or more RELATION commands.
- MODEL command executes a predefined MODEL object. Models can be used to aggregate data over non-hierarchical dimensions, which do not have a parent relation.
- CACHE command describes how or if the AGGREGATE function stores any data that is calculated on the fly. This decision controls how quickly all of a data of a variable will reflect run-time changes that users make to the variable data.
- AGGINDEX command describes whether or not Oracle OLAP should create indexes (composite tuples) that are needed by the MODEL command and by commands that use the ACROSS phrase. This is an issue only when the variable has a composite dimension.
- END command indicates the end of an aggregation map. Depending upon how you add contents to an aggmap object, you may not need to include this command explicitly.

Note: Both the CACHE and AGGINDEX commands have default settings. If these default settings are appropriate for your application, then you can omit these commands from your aggregation map. Be sure to read the topics in the Oracle9i OLAP DML Reference help for each of these commands to determine whether or not you need to use them.

Example 12–5 Simple Aggregation Map

The following is a simple aggregation map in which the data across all dimensions is precalculated using the SUM operator. Note that the body of the aggregation map begins with an AGGMAP command and ends with an END command. The RELATION

commands are listed in the order the dimensions appear in the aggmap object definition.

```
DEFINE GPCT.AGGMAP AGGMAP
LD Aggregation map for sales, units, quota, costs
AGGMAP
RELATION geography.parentrel
RELATION product.parentrel
RELATION geography.parentrel
RELATION time.parentrel
END
```

How to Compile an Aggregation Map

After you have created the aggregation map, you should compile and save it. This step is important for aggregation performed at run-time using the `AGGREGATE` function. Unless the compiled version of the aggregation map has been saved, the aggregation map will be recompiled by each session that uses it.

If you use the `FUNCDATA` argument to the `AGGREGATE` command, then the aggregation map is automatically compiled. For example, these commands will precalculate aggregate data and save a compiled copy of the aggregation map for runtime aggregation.

```
AGGREGATE sales USING gpct.aggmap FUNCDATA
UPDATE
COMMIT
```

Alternatively, you can compile the aggregation map explicitly with the `COMPILE` command. Explicitly compiling an aggregation map is also useful for finding syntax errors in the aggregation map before attempting to use it to generate data.

The following commands create and save the compiled version of the `sales.agg` aggregation map.

```
COMPILE gpct.aggmap
UPDATE
COMMIT
```

Important: If some of the data will be calculated on the fly, then you must compile and save the aggregation map *after* executing the `AGGREGATE` command.

Compiling an aggregation map can take a significant amount of time. If you fail to compile the aggregation map, the AGGREGATE function will automatically compile it in order to get the information that is needed to perform calculation on the fly. If this happens, query performance will suffer. Every time a user queries the workspace for the first time, the AGGREGATE function must compile the aggregation map before it can calculate the data. If 100 users query the same workspace, the aggregation map will be compiled 100 times. If you precompile the aggregation map and save it in the analytic workspace, then it is a task that is done once as part of the build process. If you leave the compilation to be done as a result of user queries, then it is a task that will be repeated for every user.

Aggregating Multiple Variables with a Single Command

You can use one AGGREGATE command to aggregate data for more than one variable, as long as the following conditions are true:

- All of the variables have identical dimensionality, which means that every variable definition has the same dimensions in the same order.
- You can use the same aggregation map for all of the variables. This means you will be pre-calculating the same levels of data for every variable. Therefore, you must be sure that your users tend to query the same levels of data for each variable.

Example 12–6 Variables That Can Be Aggregated with One Command

Suppose your workspace contains the following named composite and variable definitions:

```
DEFINE PROD.GEOG.CHAN COMPOSITE <PRODUCT GEOGRAPHY CHANNEL>

DEFINE SALES DECIMAL <TIME PROD.GEOG.CHAN <PRODUCT GEOGRAPHY CHANNEL>>
DEFINE UNITS INTEGER <TIME PROD.GEOG.CHAN <PRODUCT GEOGRAPHY CHANNEL>>
DEFINE PROJECTED_SALES DECIMAL <TIME PROD.GEOG.CHAN <PRODUCT GEOGRAPHY CHANNEL>>
```

Because these variables have identical dimensionality, you can use one AGGREGATE command to aggregate the data for all three variables.

Suppose you have defined an aggregation map named sales.agg. You would use the following command to aggregate data for all three variables:

```
AGGREGATE sales units projected_sales USING sales.agg
```

Example 12–7 Variables That Cannot Be Aggregated with One Command

Suppose your workspace contains the following definitions for a named composite and three variables:

```
DEFINE PROD.GEOG.CHAN COMPOSITE <PRODUCT, GEOGRAPHY, CHANNEL>

DEFINE SALES DECIMAL <TIME PROD.GEOG.CHAN <PRODUCT, GEOGRAPHY, CHANNEL>>
DEFINE UNITS INTEGER <TIME SPARSE <PRODUCT, GEOGRAPHY, CHANNEL>>
DEFINE PROJECTED_SALES DECIMAL <TIME SPARSE <PRODUCT, GEOGRAPHY>>
```

The following comparisons explain how the dimensionality is different for each variable:

- The `sales` variable uses a named composite, `prod.geog.chan`, whose base dimensions are `product`, `geography`, and `channel`.
- The `units` variable uses an unnamed composite, whose base dimensions are `product`, `geography`, and `channel`. Even though the unnamed composite has the same dimensions in the same order as the named composite, Oracle OLAP considers the named composite and the unnamed composite to be two different workspace objects. Therefore, `sales` and `units` do not have the same dimensionality.
- The `project_sales` variable also has an unnamed composite, whose base dimensions are `product` and `geography`. However, it is not identical to the unnamed composite that the `units` variable uses, because it does not include the `channel` dimension.

Because the dimensionality for each variable is different, you will have to define a different aggregation map to aggregate data for each variable. Therefore, you will have to use a different `AGGREGATE` command for each variable.

About the RELATION Command

The `RELATION` command has the following basic syntax:

```
RELATION parent-rel [PRECOMPUTE (limit-phrase)] [OPERATOR opvar]
```

An aggregation map should have one `RELATION` command for each hierarchical dimension in the definition of the variable. To promote the best possible performance, list the `RELATION` commands in the same order as they appear in the variable definition. This order indicates the way the data is stored, from fastest varying dimension to slowest varying dimension as described in "How Variable Data Is Stored" on page 3-17. When aggregating the data, it is much more efficient

to aggregate the fastest varying dimension first and the slowest varying dimension last.

For example, if the `sales` variable is dimensioned by `time` and the `prod.geog.chan` composite like this:

```
<time prod.geog.chan <product, geography, channel>>
```

Then the first `RELATION` command should be for `time`, the second for `product`, the third for `geography`, and the fourth for `channel`.

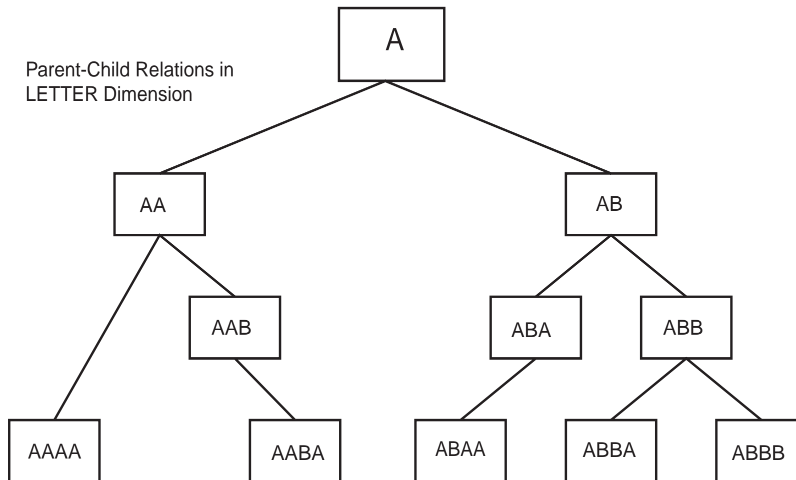
Example 12–8 Aggregating with SUM or MAX

The following examples use the `letter` dimension, the `letter.letter` parent relation, and the `units` variable.

```
LETTER         LETTER.LETTER    UNITS
-------------- ------------- -------------
a              NA                       NA
aa             a                        NA
ab             a                        NA
aab            aa                       NA
aba            ab                       NA
abb            ab                       NA
aaaa           aa                        1
aaba           aab                       2
abaa           aba                       1
abbb           abb                       1
abba           abb                       1
```

The following illustration shows the relations defined by `letter.letter`.

Figure 12–1 Parent-Child Relationships in the LETTER Dimension



`LETTER.AGGMAP` uses `SUM` to calculate the value of `aa`.

```
DEFINE LETTER.AGGMAP AGGMAP
AGGMAP
RELATION letter.letter PRECOMPUTE ('aa')
END
```

When the data is aggregated, `aa` has a value of `3`:

```
aa = (aab + aaaa) = (aaba + aaaa) = (2 + 1) = 3
```

Note that although `aab` is the parent of `aaba` and the child of `aa`, its value is not stored as the result of this calculation.

Specifying an Aggregation Method

The aggregation method for each dimension is specified in the `RELATION` command. The default aggregation method is `SUM`, which adds the values of the

child cells and stores the total in the parent cell. However, there are other aggregation methods that you can use:

Sum (SUM)
Scaled Sum (SSUM)
Weighted Sum (WSUM)
Average (AVERAGE)
Hierarchical Average (HAVERAGE)
Weighted Average (WAVERAGE)
Hierarchical Weighted Average (HWAVERAGE)
Maximum (MAX)
Minimum (MIN)
First (FIRST)
Hierarchical First (HFIRST)
Last (LAST)
Hierarchical Last (HLAST)
And (AND)
Or (OR)

These aggregation methods are arguments to the RELATION command. For descriptions of these methods, refer to the RELATION command entry in Oracle9i OLAP DML Reference help. Do not confuse the RELATION aggregation methods with the DML aggregation functions.

Example 12–9 Specifying the Aggregation Method

The OPERATOR keyword in the following RELATION command changes the method of aggregation from the default SUM to MAX.

```
RELATION letter.letter PRECOMPUTE ('aa') OPERATOR MAX
```

When the data is aggregated with the modified aggregation map, aa has a value of 2, because 2 is the largest value contributing to aa, as shown in Figure 12–1, "Parent-Child Relationships in the LETTER Dimension".

Example 12–10 Using a Weighted Variable

Several aggregation methods use weighted variables: WSUM, WAVERAGE, and HWAVERAGE. You must first define a weighted variable, then specify it in the RELATION command using the ARGS WEIGHTBY argument.

The following aggregation map uses the weights defined in variable `letter.weights` to calculate the value of `aa`.

```
DEFINE LETTER.AGGMAP AGGMAP
AGGMAP
RELATION letter.letter PRECOMPUTE ('aa') OPERATOR WSUM -
     ARGS WEIGHTBY letter.weights
END
```

The output from this `REPORT` command shows the aggregation.

```
report down letter letter.weights units


LETTER          LETTER.LETTER  LETTER.WEIGHTS      UNITS
-------------- --------------- --------------- ---------------
a              NA                           NA              NA
aa             a                            NA               7
ab             a                            NA              NA
aab            aa                           NA              NA
aba            ab                           NA              NA
abb            ab                           NA              NA
aaaa           aa                            5               1
aaba           aab                          NA               2
abaa           aba                          NA               1
abbb           abb                          NA               1
abba           abb                          NA               1
```

The value of aa in the `units` variable is calculated in this way:

```
aa = ((5 * aaaa) + aab) = ((5*aaaa) + aaba) = (5*1) + 2 = 7
```

Selecting Data For Aggregation

The `PRECOMPUTE` clause limits the data that is aggregated by the `AGGREGATE` command. In its simplest form, the `PRECOMPUTE` clause is like a `LIMIT` *`dimension`* `TO` command. Notice that the default limit is on the dimension, which is not explicitly named in the `RELATION` command.

For example, this `LIMIT` command selects the `AUDIODIV`, `VIDEODIV`, and `ACCDIV` values of the `product` dimension:

```
limit product to 'audiodiv' 'videodiv' 'accdiv'
```

The equivalent RELATION command looks like this:

```
RELATION product.parentrel PRECOMPUTE ('AUDIODIV' 'VIDEODIV' 'ACCDIV')
```

Since these values are all at the same level of the product STANDARD hierarchy (L2), this LIMIT command yields the same results:

```
limit product to product.levelrel 'L2'
```

This is the equivalent RELATION command:

```
RELATION product.parentrel PRECOMPUTE (product.levelrel 'L2')
```

The TO clause may not always produce the results you want. To use the other selection clauses (such as KEEP, REMOVE, and COMPLEMENT), you must explicitly call the LIMIT function.

```
RELATION product.parentrel PRECOMPUTE (limit(product complement 'TOTALPROD'))
```

Example 12–11 Aggregation Map with PRECOMPUTE Clauses

This aggregation map uses PRECOMPUTE clauses to limit the data that is aggregated by the AGGREGATE command.

```
DEFINE GPCT.AGGMAP AGGMAP
LD Aggregation map for sales, units, quota, costs
AGGMAP
RELATION geography.parentrel PRECOMPUTE (geography.levelrel 'L3')
RELATION product.parentrel PRECOMPUTE (limit(product complement 'TOTALPROD'))
RELATION channel.parentrel
RELATION time.parentrel PRECOMPUTE (time ne '2001')
END
```

Caching Runtime Aggregates

The CACHE command in an aggregation map determines whether data that is calculated on the fly is available for the duration of a session. By default, the data must be recalculated each time it is queried. The user will experience faster querying time if the data is cached and simply retrieved for subsequent queries, however, maintaining a cache can have unwanted side-effects.

If users alter the data during their sessions (such as when running forecasts and what-if analysis), then data that was aggregated previously will not reflect the changes in the data. Having the data out of synchronization in this way means that users will view inaccurate data. Do not maintain a cache if users alter the data during their sessions.

If users have write access to the analytic workspace, then the runtime calculations will be saved along with other changes if a user issues UPDATE and COMMIT commands. This defeats the purpose of runtime aggregation, which is to save storage space.

If users can save their analytic workspaces, then create a cache using a CACHE SESSION command. If they cannot save their workspaces, then you can use either CACHE SESSION or CACHE STORE.

The effectiveness of a cache is tracked in the V$AW_CALC dynamic performance view. See the *Oracle9i OLAP User's Guide* for information about querying this view.

Aggregating Non-Hierarchical Data

Some dimensions, such as line items, do not have a hierarchical structure. Instead, individual line items are calculated, sometimes with complex formulas, from one or more other line items or workspace objects. Models are needed to solve the data over this type of a dimension.

To execute a model, you include a MODEL command within the aggmap. It has the following basic syntax:

```
MODEL modelname [PRECOMPUTE ALL|NA]
```

Where:

modelname is the name of an existing MODEL object that calculates values for one or more dimensions of the aggregation map.

PRECOMPUTE ALL indicates that the AGGREGATE command will execute the model as a data maintenance step. Any RELATION or MODEL commands that precede it in the aggregation map must also be specified as PRECOMPUTE ALL. However, any RELATION or MODEL commands that follow it in the aggregation map can either be specified as PRECOMPUTE ALL or PRECOMPUTE NA.

PRECOMPUTE NA indicates that the AGGREGATE function will execute the model at runtime. The following conditions must be met for runtime execution:

- All RELATION commands in the aggmap must appear before the MODEL command specified as PRECOMPUTE NA.
- Any additional MODEL commands that follow it must also be specified as PRECOMPUTE NA.
- The model cannot solve simultaneous equations or time series (such as LEAD and LAG functions).

- The model cannot reference an object that invokes the AGGREGATE function. For example, the model can contain an equation such as TAX=PROFIT*RATE where RATE is a variable or formula. However, RATE cannot require runtime aggregation.

See Also: Chapter 8, "Working with Models"

Example 12–12 Solving a Model in an Aggregation

This example uses the budget variable:

```
DEFINE BUDGET VARIABLE DECIMAL <LINE TIME>
LD Budgeted $ Financial
```

The time dimension has two hierarchies (STANDARD and YTD) and a parent relation named time.parentrel as follows:

```
               -----TIME.PARENTREL------
               ----TIME.HIERARCHIES-----
TIME             STANDARD       YTD
-------------- ------------ ------------
LAST.YTD       NA           NA
CURRENT.YTD    NA           NA
JAN01          Q1.01        LAST.YTD
FEB01          Q1.01        LAST.YTD
MAR01          Q1.01        LAST.YTD
APR01          Q2.01        LAST.YTD
MAY01          Q2.01        LAST.YTD
JUN01          Q2.01        LAST.YTD
JUL01          Q3.01        LAST.YTD
AUG01          Q3.01        LAST.YTD
SEP01          Q3.01        LAST.YTD
OCT01          Q4.01        LAST.YTD
NOV01          Q4.01        LAST.YTD
DEC01          Q4.01        LAST.YTD
JAN02          Q1.02        CURRENT.YTD
FEB02          Q1.02        CURRENT.YTD
MAR02          Q1.02        CURRENT.YTD
APR02          Q2.02        CURRENT.YTD
MAY02          Q2.02        CURRENT.YTD
Q1.01          2001         NA
Q2.01          2001         NA
Q3.01          2001         NA
Q4.01          2001         NA
Q1.02          2002         NA
```

```
Q2.02          2002         NA
2001           NA           NA
2002           NA           NA
```

The relationships among line items are defined in the following model.

```
DEFINE INCOME.BUDGET MODEL
MODEL
dimension line time
opr.income = gross.margin - marketing
gross.margin = revenue - cogs
revenue = lag(revenue, 12, time) * 1.02
cogs = lag(cogs, 1, time) * 1.01
marketing = lag(opr.income, 1, time) * 0.20
END
```

The following aggregation map pre-aggregates all of the data. Note that all of the data must be pre-aggregated because the model includes both `LAG` functions and a simultaneous equation.

```
DEFINE BUDGET.AGGMAP1 AGGMAP
AGGMAP
MODEL income.budget
RELATION time.parentrel
END
```

How to Generate Precalculated Data

Typically, you will general precalculated aggregates in a batch window as part of maintaining the data in your database. If you wish, you can use Job Manager to schedule batch jobs in Oracle Enterprise Manager, as described in the *Oracle9i OLAP User's Guide.*

The `AGGREGATE` command aggregates the data for one or more variables according to the specifications provided in the aggregation map. The basic syntax of the `AGGREGATE` command is:

```
AGGREGATE variables USING aggmap
```

Where:

`variables` is the name of one or more variables.

`aggmap` is the name of the aggregation map.

Example 12–13 Precalculating Data in a Batch Job

Your batch job should include commands like the following:

```
ALLSTAT
POUTFILEUNIT=FILEOPEN('userfiles/progress.txt' WRITE)
AGGREGATE sales units USING gpct.aggmap
UPDATE
COMMIT
FILECLOSE POUTFILEUNIT
```

Effects of Dimension Status

The RELATION command only aggregates those source data values (that is, those values that are loaded into the analytic workspace and used as the basis of aggregation) that are in status. The parent values are calculated regardless of whether they are in status or not. For example, if only JAN01, FEB01, and MAR01 are in status for the time dimension, then Q1.01 will be calculated (but no other quarters), and 2001 will be calculated (but no other years) using only Q1.01 as input since the other quarters are NA.

This can be useful when you want to aggregate just the new data in your analytic workspace. However, you must exercise some care, as described in "Performing Partial Aggregations" on page 12-27.

Monitoring Progress

You can monitor the progress of an aggregation by setting the POUTFILEUNIT option. You can use the OUTFILEUNIT option or the OUTFILE function to set the value of POUTFILEUNIT.

This command sets POUTFILEUNIT to the file unit number of the current outfile, which is usually the screen:

```
POUTFILEUNIT=OUTFILEUNIT
```

This command opens a file named progress.txt in the userfiles directory alias, and sets POUTFILEUNIT to the file unit number of progress.txt:

```
POUTFILEUNIT=FILEOPEN('userfiles/progress.txt' WRITE)
```

When the aggregation is complete, you must close the file with a FILECLOSE command.

How to Calculate Data at Runtime

The AGGREGATE function calculates the complement of the data specified in the PRECOMPUTE clause of the RELATION command. It returns those values that are currently in status.

For example, if you are using an aggregation map that contains this RELATION command:

```
RELATION letter.letter PRECOMPUTE ('aa')
```

Then the AGGREGATE function calculates all aggregations *except* aa, as shown here.

```
REPORT AGGREGATE(units USING letter.aggmap)

                  AGGREGATE(UNITS
LETTER          USING LETTER.AGGMAP)
-------------- --------------------
a                                 3
aa                               NA
ab                                3
aab                               2
aba                               1
abb                               2
aaaa                              1
aaba                              2
abaa                              1
abbb                              1
abba                              1
```

Setting Up Calculation on the Fly

If you want to calculate some data on the fly, you need to perform the following steps:

1. Decide which data should be pre-calculated and which data should be calculated on the fly.
2. Define an aggregation map that contains the PRECOMPUTE keyword in one or more RELATION or MODEL commands. It may also contain a CACHE command if the default value is not appropriate.
3. Use the AGGREGATE command with the aggregation map to pre-calculate the data that will be stored on disk.

4. Compile the aggregation map *after* executing the AGGREGATE command, as explained in "How to Compile an Aggregation Map" on page 12-10.
5. Add the $NATRIGGER property to the variables that use the aggregation map, so that NAs in queried data will cause the AGGREGATE function to execute.

Adding the $NATRIGGER Property to a Variable

Instead of specifying the AGGREGATE function in every command that you want to return aggregate data, you can add a property to the variable so that the AGGREGATE function is executed automatically.

An $NATRIGGER property on a variable indicates that NA values in the queried data will cause a particular action to take place. To trigger an aggregation, a call to the AGGREGATE function is the value assigned to the $NATRIGGER property.

The following commands add the $NATRIGGER property to the sales variable, so that unsolved data will be aggregated using the sales.aggmap aggregation map:

```
CONSIDER sales
PROPERTY '$NATRIGGER' 'AGGREGATE(sales USING sales.aggmap)'
```

Creating Custom Aggregates

Most aggregates are defined with a parent relation that identifies the parent-child relationships within the dimension. However, users may wish to create their own aggregates at runtime, perhaps for forecasting or what-if analysis, or just because they want to view the data in an unforeseen way. This is the process by which a custom aggregate is created:

1. Create a dimension value for the custom aggregate. The following command adds 'bb' to the letter dimension:

   ```
   maintain letter add 'bb'
   ```

2. Create a MODEL object that contains an AGGREGATION function, which associates child dimension values with the new dimension value. The following model identifies bb as the parent of aab and aba. Note that the parent dimension value (in this case, bb) cannot already be defined as a parent in the parent relation (letter.letter).

   ```
   DEFINE LETTER.MODEL MODEL
   MODEL
   DIMENSION letter
   bb=AGGREGATION('aab' 'aba')
   ```

3. Execute an AGGMAP ADD command to append the model to an existing AGGMAP object.

```
AGGMAP ADD letter.model TO letter.aggmap
```

The aggregation map from Example 12–8 now looks like this:

```
DEFINE LETTER.AGGMAP AGGMAP
AGGMAP
RELATION letter.letter PRECOMPUTE ('aa')
END
AGGMAP ADD letter.model
```

4. The model is executed only by the AGGREGATE function like the one shown here; the AGGREGATE command ignores it.

```
REPORT AGGREGATE(units USING letter.aggmap)
```

5. If you wish to remove the model from the aggregation map during a session, use the AGGMAP REMOVE command.

Important: The AGGMAP ADD command is automatically removed from an aggmap object at the end of a session.

Balancing Precalculated and Runtime Aggregation

Using AGGREGATE, all of the following strategies are possible. You can:

- Pre-aggregate all of the data. This means that all of the data for the variable will be aggregated and stored in the database. This is likely to result in relatively slow build performance and extremely fast user query performance.
- Calculate all of the data on the fly, that is, at run-time. In this case, you eliminate aggregation from the build process, which means that build performance will be very fast; it will be reduced to the time that it takes to load data into the workspace. However, user query performance will suffer greatly.
- Pre-aggregate some of the data and calculate the remainder on the fly.

Good performance is a matter of trade-offs. Therefore, one of the most effective steps you can take to achieve overall good performance is to balance the amount of the data that you aggregate and store in an analytic workspace with the amount of data that you specify for calculation on the fly.

A typical strategy is **skip-level aggregation**: that is, select one or two of a variable's dimensions and pre-aggregate every other level in those dimension's hierarchies. If you know which levels are queried most often by users, you should pre-calculate those levels of data.

Example 12–14 Calculating Data Using the Skip-Level Approach

Suppose you want to aggregate `sales` data. The `sales` variable is dimensioned by `geography`, `product`, `channel`, and `time`.

First, consider the hierarchy for each dimension. How many levels does each hierarchy have? What levels of data do users typically query? If you are designing a new workspace, what levels of data do your users plan to query?

Suppose you learn the following information about how users tend to query `sales` data for the `time` hierarchy:

Time Level Names	Descriptive Level Name	Examples of Dimension Values	Do users query this level often?
`L1`	Year	`YEAR99`, `YEAR00`	yes
`L2`	Quarter	`Q3.99`, `Q3.99`, `Q1.00`	yes
`L3`	Month	`JAN99`, `DEC00`	yes

The following information shows how your users tend to query `sales` data for the `geography` hierarchy:

Geography Level Names	Descriptive Level Name	Examples of Dimension Values	Do users query this level often?
`L1`	World	`WORLD`	yes
`L2`	Continent	`EUROPE`, `AMERICAS`	no
`L3`	Country	`HUNGARY`, `SPAIN`	yes
`L4`	City	`BUDAPEST`, `MADRID`	yes

The following information shows how your users tend to query sales data for the product dimension hierarchy:

Product Level Names	Descriptive Level Name	Examples of Dimension Values	Do users query this level often?
L1	All Products	TOTALPROD	yes
L2	Division	AUDIODIV, VIDEODIV	yes
L3	Category	TV, VCR	yes
L4	Product	TUNER, CDPLAYER	yes

Using this information about how users query data, you should use the following strategy for aggregation:

- Fully aggregate time and product because all levels are queried frequently.
- For the geography dimension, aggregate data for L1 (World) and L3 (Country) because they are queried frequently. However, L2 is queried less often and so can be calculated on the fly.

The lowest level of data was loaded into the analytic workspace. The aggregate data is calculated from this source data.

Therefore, the contents of the aggregation map might look like the following:

```
RELATION time.parentrel
RELATION geography.parentrel PRECOMPUTE (geog.leveldim 'L3' 'L1')
RELATION product.parentrel
```

Selecting Dimensions for Runtime Calculation

Use a skip-level approach for only one or two dimensions. You should use the skip-level approach for half or fewer of the dimensions in a variable definition. For example, if there are three dimensions, then you can use the skip-level approach for one dimension; if there are four or more dimensions, then you can use the skip-level approach for two dimensions.

The dimensions that are the best candidates for skip-level aggregation are the dimensions whose hierarchies have many levels.

If possible, choose a dimension that is either fastest- or intermediate-varying in the variable dimension. Performance of calculation on the fly will always be best for dimensions in this position.

Selecting Levels for Runtime Calculation

Skip every other level in a dimension hierarchy, and avoid skipping more than two levels that are adjacent to each other. For example, if a hierarchy has seven levels, you might skip `L2`, `L4`, and `L6`. That means you would precalculate `L1`, `L3`, and `L5`. (The detail-level data is at `L7`.) Take into consideration how frequently a level is queried, as demonstrated in Example 12–14. Users will experience the best performance if you pre-aggregate the data most frequently queried, and aggregate on the fly the data that is requested occasionally.

Do not skip adjacent levels. For example, if you skipped `L2`, `L3`, `L4`, and `L5`, then a query for `L2` data would require `AGGREGATE` to calculate `L5`, then aggregate that data up to `L4`, then up to `L3`, and finally to `L2`. Alternatively, if you skip `L2`, `L4`, and L6, then a query for `L2` data requires `AGGREGATE` to aggregate data only from `L3`.

The one exception to this rule is when each level has very few children per parent. When this is true for every adjacent level that you want to skip, then you can skip two or more adjacent levels.

Performing Partial Aggregations

Maintenance of an analytic workspace must usually be done within a restrictive batch window. For this reason, many DBAs perform partial aggregations rather than full aggregations each time they refresh the data. When all of the data is pre-aggregated, this does not present a problem. However, when partial aggregations are performed on data that uses both pre-aggregation and runtime aggregation, then steps must be taken to ensure that the results are correct. Errors in the data occur when the status list generated by the `PRECOMPUTE` keyword is outdated.

The `PRECOMPUTE` clause produces a status list that:

- Tells the `AGGREGATE` command which data should be pre-calculated, and
- Tells the `AGGREGATE` function what the `AGGREGATE` command has done

If you never use the `AGGREGATE` command with the `AGGREGATE` function, you do not need the information.

Aggregation Changes That Cause Problems

You should read this information to address the following circumstances:

- Incremental data loading: You have already built your analytic workspace and are now loading new data on a regular basis. You make a change to at least one `PRECOMPUTE` clause in a `RELATION` command in an aggregation map.
- Using data-dependent `PRECOMPUTE` clauses: When you use the `PRECOMPUTE` keyword in an aggregation map, that `PRECOMPUTE` clause can be data-dependent instead of simply identifying dimension values or levels.
- Changing a hierarchy: If you make a change to a dimension's hierarchy after you have already aggregated data, then you will need to aggregate all of the data again. There is a procedure you can use, in some cases, to reduce the time it takes to re-aggregate the data in your analytic workspace.

Incremental Data Loading

Incremental data loading refers to the process of loading new input data into an existing analytic workspace and then aggregating that data. This usually happens on a regular basis, whether it is on a monthly, weekly, or even daily basis.

For example, suppose you design a new analytic workspace. It contains two variables: `sales` and `units`. Suppose that when you build the analytic workspace for the first time, you have input data for one year for both variables. Because `sales` and `units` contain exactly the same dimensions in exactly the same order in their definitions, you define one aggregation map that will be shared by both `sales` and `units`. You load that input data into the analytic workspace, then use the `AGGREGATE` command to roll up that input data.

You know that you will be getting new input data for `sales` and `units` on the first day of every month. For example, suppose it is March 1. On this day, you expect to receive the sales data and units data for the previous month of February. Your responsibility is to load the February data into the existing analytic workspace and aggregate that input data. This is an incremental data load. The next incremental data load will take place on April 1, and so on.

Typically, when you aggregate this new data, you will use a `LIMIT` command to ensure that only the new input data will be aggregated. For example, to aggregate only the new input data that you have loaded for February, you might use the following commands:

```
LIMIT month TO 'FEB99'
AGGREGATE sales units USING salesunits.aggmap
```

This is acceptable as long as you do not change any of the PRECOMPUTE clauses in the aggregation map. If you do, then you must pre-aggregate all of the data.

Problem: PRECOMPUTE Status List Is Inaccurate

If you change a PRECOMPUTE clause, then the status list will change. This means that although the data that is produced by the AGGREGATE command after you change the PRECOMPUTE clause will be correct, Oracle OLAP may not be able to return the data that is requested by a user using the AGGREGATE function. The status list might indicate that a value has already been calculated when in fact it has not.

Solution: Regenerate the PRECOMPUTE Status List

If you make any changes to any PRECOMPUTE clause in one or more RELATION commands in an aggregation map, then you must pre-aggregate all of the data. Otherwise, the AGGREGATE function will use a PRECOMPUTE status list that is out of synchronization with the data, and thus may not generate all of the required values.

Use the following procedure to be sure the data will be aggregated correctly:

1. Make sure that you have finished making any changes that you want to make to the PRECOMPUTE clauses in your aggregation map.
2. Load the incremental input data.
3. Set the current status of all dimensions to ALL. (You can use one ALLSTAT command, or a LIMIT TO ALL command for every dimension in the aggregation map.)
4. Execute the AGGREGATE command.
5. Recompile the aggregation map. (Alternatively, you can use the FUNCDATA keyword when you execute the AGGREGATE command in Step 4.)

Using a Data-Dependent PRECOMPUTE Clause

The clause that follows the PRECOMPUTE keyword is like a LIMIT command. You have the flexibility to specify the limit expression using the values of the data. For example, you can specify the five areas with the lowest sales figures in a time period. The RELATION command might look like this:

```
RELATION geography.parentrel PRECOMPUTE (BOTTOM 5 BASEDON sales)
```

Problem: Values of the Limit Clause Vary With Each Data Update

Data-dependent limit expressions can vary in their results. In other words, the "bottom five" areas in the analytic workspace that you build in February will not necessarily be the same "bottom five" areas after performing an incremental data load in March. Furthermore, the "bottom five" areas in your March will not necessarily be the same "bottom five" areas after the April incremental data load.

In this situation, the `PRECOMPUTE` status list is out of synchronization, and the `AGGREGATE` function may not calculate a needed value because the status list indicates that it was precomputed.

Solution: Maintain a Valueset

Instead of using a data-dependent `PRECOMPUTE` clause, you can either:

- List specific values, ***or***
- Create a valueset that stores specific values

As you load and aggregate incremental data over the course of time, the status list that is generated by the `PRECOMPUTE` keyword remains constant when you use one of these methods. However, the five stores in the limit expression or valueset remains the same, regardless of whether or not they still represent the stores with the lowest sales figures.

To keep the limit phrase current, take the following steps:

1. Recompute the limit expression each time you load new data.
2. Change the valueset when the results of your computation are different.
3. Perform a full aggregation of the affected variables.
4. Recompile the aggregation map that is used by the `AGGREGATE` function.

Refer to "Incremental Data Loading" on page 12-28 for the general guidelines you should follow.

If you have changed the input data or your hierarchies, then replace any data that has been aggregated with `NA` values. These are the steps that you might take: Limit the dimensions to the input data, create a new variable, copy the data from the original variable to the new variable, delete the original variable, and rename the new variable to the name of the original variable.

Example 12–15 Listing the Dimension Values

Instead of using data-dependent PRECOMPUTE clauses, use specific dimension values in the PRECOMPUTE clause. After loading the data, issue a data-dependent LIMIT command to identify the dimension values. Then list those values in the PRECOMPUTE clause. For example,

```
LIMIT time TO '2001'
LIMIT channel TO 'TOTALCHANNEL'
LIMIT product TO 'TOTALPROD'
LIMIT geography TO BOTTOM 5 BASED ON sales

STATUS geography
The current status of GEOGRAPHY is:
BOGOTA, BORDEAUX, EDINBURGH, KYOTO, BRUSSELS
```

You would then change the PRECOMPUTE clause to list these areas:

```
RELATION geography.parentrel PRECOMPUTE ('BOGOTA' 'BORDEAUX' 'EDINBURGH' -
    'KYOTO' 'BRUSSELS')
```

If you want to use data-dependent PRECOMPUTE clauses, create and use a valueset with the PRECOMPUTE clause.

Example 12–16 Using a Valueset

A valueset can be used to store a list of values. For example, the following commands create a valueset for the geography dimension. After performing an incremental update, you would need to update the valueset, but you would not need to edit the aggregation map.

The following commands create a valueset for geography:

```
DEFINE lowsales.geog VALUESET geography
LIMIT time TO '2001'
LIMIT channel TO 'TOTALCHANNEL'
LIMIT product TO 'TOTALPROD'
LIMIT lowsales.geog TO BOTTOM 5 BASED ON sales
```

The VALUES function returns the status list of the valueset:

```
SHOW VALUES(lowsales.geog)

BOGOTA
BORDEAUX
EDINBURGH
BRUSSELS
KYOTO
```

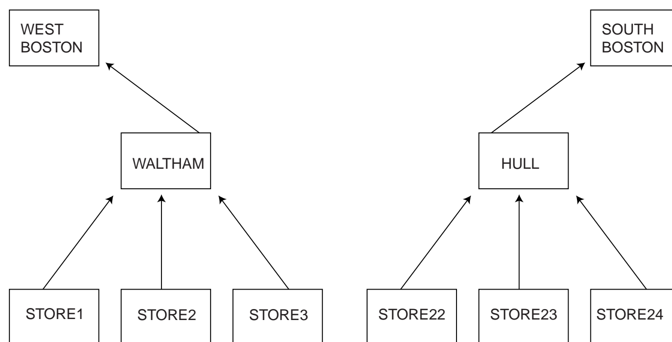
This RELATION command uses the valueset:

```
RELATION geography.parentrel PRECOMPUTE (lowsales.geog)
```

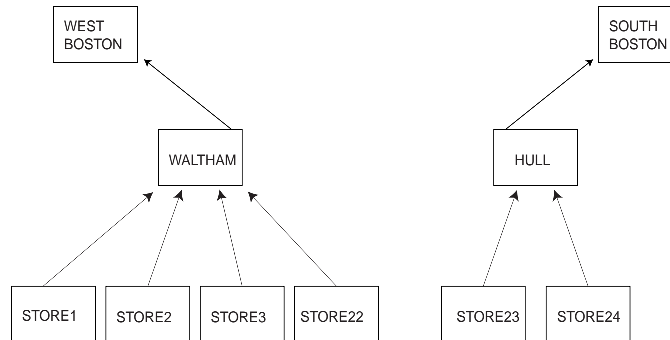
Changing a Hierarchy

Once you have defined a hierarchy and you have aggregated data, if you move one or more dimension values to a different parent in the hierarchy, then you have changed the hierarchy.

For example, suppose your geography hierarchy has input data for stores. The store data rolls up into cities. The cities roll up into regions, and so on.



You define your dimensions and variables. You define the hierarchies for your dimensions. You load data and roll it up. Several months later, after you have loaded and rolled up incremental data, one of the stores changes location. For example, STORE22 closes its location in Hull, Massachusetts and then reopens at a new location in Waltham, Massachusetts. Therefore, STORE22 now is part of the WEST BOSTON region instead of SOUTH BOSTON region.



Therefore, you must move the STORE22 dimension value so that its data will roll up to different dimension values in the higher levels of the hierarchy. For example, you must move STORE22 from the HULL path to the WALTHAM path.

When you move one or more dimension values so that their data rolls up in a different path in the hierarchy, you have changed the hierarchy.

Problem: Previously Aggregated Data is Incorrect

Suppose that you receive the most recent month's worth of data for STORE22. You load that data and aggregate it.

Today you find out that last month the store moved to a new city, as well as a new region. This means that you have already aggregated the STORE22 data into HULL, when the STORE22 data now should be aggregated into WALTHAM.

The problem is that you not only need to change the hierarchy, but you need to correct the data so that the STORE22 data aggregates into WALTHAM instead of HULL.

Solution: Re-Aggregate Changed Branches

When you change a hierarchy, you can re-aggregate the data in the analytic workspace (after you have changed the hierarchy) in one of two ways:

- Perform a full aggregation. This is the best alternative if you make significant changes to a hierarchy.
- Perform a partial aggregation for the dimension value that has moved, as well as a previous sibling of that dimension value. This method is acceptable for very small changes to the hierarchy.

The advantage of a partial aggregation is that it takes a shorter period of time to complete than a full aggregation. However, the advantage of performing a full rollup is that you know the results will be correct.

Therefore, if you move one or two dimension values in your hierarchy, and you have a small window of time to roll up the analytic workspace, you can perform a partial aggregation; otherwise, perform a full aggregation.

How to Aggregate Branches of a Hierarchy

Follow these steps to aggregate the data for the former parents and the current parents of the dimension value that moved in the hierarchy.

1. Identify the dimension value (or group of dimension values) that has moved in the hierarchy. For example, `STORE22` is the dimension value whose data now aggregates to `WALTHAM` instead of `HULL`.
2. Identify a previous sibling of the dimension value that has moved. (If more than one dimension has moved, you must identify a sibling for each one.) For example, `STORE22` was previously grouped with `STORE23` and `STORE24`; either one qualifies as a previous sibling of `STORE22`.
3. Limit the current status of the dimension to the dimension value that has moved and its previous sibling. For example, use the following command to limit the `geography` dimension to `STORE22` and `STORE23`.

```
LIMIT time TO 'STORE22' 'STORE23'
```

4. Aggregate the variable's data. For example, use the following command to aggregate the `sales` variable.

```
AGGREGATE sales USING sales.agg
```

By identifying the dimension value that has moved, you can recalculate its new ancestors (such as `WALTHAM`). By identifying a previous sibling of the dimension value that has moved, you can recalculated its previous ancestors (such as `HULL`).

Combining AGGREGATE with Forecasts and Programs

You will need to use multiple aggregation maps for a single variable when you use alternative ways to pre-aggregate data (in addition to the `AGGREGATE command`) over one or more dimensions. These alternative ways to aggregate data can include:

- A forecast
- An OLAP DML program

For example, suppose the sales variable is dimensioned by geography, product, channel, and time. You If you aggregate some of a variable's data with AGGREGATE, and you aggregate other data for that same variable with a forecast or DML program, then you need to take extra steps to make sure all of the aggregated data will be correct.

Important: You should not use the AGGREGATE function with multiple aggregation maps unless you feel comfortable answering the following question:

When the aggregation map is compiled for use by the AGGREGATE function, does the status that results from each PRECOMPUTE clause accurately define the nodes within that dimension at which data has been pre-computed?

If you cannot answer "yes" to this question with confidence, you should not use the AGGREGATE function with multiple aggregation maps.

When to Use Multiple Aggregation Maps

Ideally, you will use the same aggregation map with both the command and the function to aggregate data for the same variable or group of variables. However, it may be necessary to use two or more aggregation map with the command and a different aggregation map with the function to assure that the results will be correct.

Problem: Different Aggregation Maps Generate Different Status Lists

The reason for using multiple aggregation maps is that each one performs a different task, and thus produces a different status list.

If the AGGREGATE command and the AGGREGATE function use the same aggregation map, then there is no problem; they will be using the same status list, because only one status list exists.

The problem occurs when you use more than one aggregation map with the AGGREGATE command for the same variable (or group of variables). Each one produces a different status list, and none of them alone may correctly identify the current status for the AGGREGATE function.

Solution: Create a Separate AGGMAP for the AGGREGATE Function

If you use more than one aggregation map to pre-calculate data, then you must:

1. Create another aggregation map for the `AGGREGATE` function, which will be used for user queries.
2. Make sure that the contents of the aggregation map for user queries combines the contents of the aggregation maps that you use to pre-calculate data. Refer to Example 12–17, "Using Multiple Aggregation Maps" for an example of how to do this.

Example 12–17 Using Multiple Aggregation Maps

If you use a forecast, you must make sure that all of the input data that is required by that forecast has been pre-calculated. Otherwise, the forecast will use incorrect or nonexistent data.

For example, suppose your forecast requires that all line items are aggregated. Using a `budget` variable that is dimensioned by `time`, `line`, `and division`, a typical approach would be to perform a complete aggregation of the `line` dimension, forecast the `time` dimension, and then aggregate the remaining dimension, `division`. Define the first aggregation map, named `forecast.agg1`, which aggregates the data needed by the forecast. It contains the following command:

```
RELATION line.parentrel
```

Define the second aggregation map, named `forecast.agg2`, which aggregates the data generated using the first aggregation map and the forecast. It contains the following command:

```
RELATION division.parentrel PRECOMPUTE ('L3')
```

Define the third aggregation map, named `forecast.agg3`, which contains the contents of the first two aggregation maps:

```
RELATION line.parentrel
RELATION division.parentrel PRECOMPUTE ('L3')
```

If your forecast is in a program named `fore.prg`, then you would use these commands to aggregate the data:

```
AGGREGATE budget USING forecast.agg1   "Aggregate over LINE
CALL fore.prg                          "Forecast over TIME
AGGREGATE budget USING forecast.agg2   "Aggregate over DIVISION
```

```
"Compile the limit map for LINE and DIVISION
COMPILE forecast.agg3

"Use the combined aggregation map for the AGGREGATE function
CONSIDER budget
PROPERTY 'NATRIGGER' 'AGGREGATE(budget USING forecast.agg3)'
```

Index

Symbols

% wildcard, 4-28
& operator, 4-30, 4-31
= command
  ACROSS keyword, 5-12
  example of, 5-12, 5-13
  introduced, 4-3, 5-3, 5-10
  saving calculations, 5-12
  with composites, 5-12
  with dimensions, 5-14
  with models, 8-5
  with QDR, 4-8, 5-14
  with relations, 5-14
  with variables, 5-11, 5-12
  with variables using composites, 5-12, 5-13
= operator, *See* = command
_ wildcard, 4-28

A

ABS function, 4-24, 4-25
ACROSS phrase
  used when reading files, 11-17
AGGINDEX command
  definition, 12-9
  purpose of, 12-9
AGGMAP command, 3-27, 12-7
aggmap object, *See* aggregation map
AGGMAPINFO command, 9-4
AGGREGATE command
  introduced, 12-4
  multiple variables, 12-11
AGGREGATE function
  adding as a property to the variable, 12-23
  introduced, 12-4
aggregating data
  best practice, 12-24
  for multiple variables, 12-11
  list of commands, 12-3
  methods, 12-14
  on-the-fly, 12-2
  overview, 12-2
  precomputing, 12-2
  process, 12-4
aggregation functions, NA values in, 4-33
aggregation map
  commands for allocation, 9-5
  compiling, 12-10
  creating aggmap object, 12-7
  for allocation, 9-5
  how to define, 12-7
  performance tip, 12-12
  RELATION command, 9-6, 12-12
alias
  analytic workspace, 2-8
  directory, 11-4
ALLOCATE command, 9-2, 9-4
allocating data
  introduction to, 9-2
  list of related commands, 9-4
  preparing for, 9-5
ALLOCERRLOGFORMAT command, 9-4
ALLOCERRLOGHEADER command, 9-4
ALLOCMAP command, 3-27, 9-4, 9-5
ampersand (&) operator, 4-30, 4-31

ampersand substitution
avoiding, 4-31
defined, 4-30
effect performance, 7-9
example of, 4-31
prevents compiling, 7-29
program arguments and, 7-9
QDR with, 4-10
restrictions, 8-6
using to pass arguments, 7-9
when required, 7-9
analytic workspaces
access from Java, 1-10
access from OLAP Worksheet, 1-6
access from SQL, 1-9
acquiring description of, 2-15
active workspace, 2-2
alias, 2-8
attached read-only or read/write, 2-4
attached workspace, 2-2
attaching, 2-3
committing changes, 2-9
controlling access to, 2-12
copying data into relational tables, 10-28 to 10-32
creating, 2-3
current workspace, 2-2
deleting, 2-5
detaching, 2-5
exporting, 2-14
importing, 2-14
introduction to, 1-2
list of attached, 2-2
minimizing growth of, 2-10
multiple, 2-6
name, 2-7
objects, acquiring information about, 2-15, 2-16, 2-17
objects, defining, 3-2
objects, defining in a program, 7-29
permission programs, 2-12, 2-13
populating, 5-1
populating from relational tables, 10-3 to 10-20
reorganizing, 2-10
retrieving name of, 2-2
saving changes to, 2-8
security, 2-12
sharing across sessions, 2-4
updating, 2-9
waiting for access, 2-5
AND operator, 4-21, 4-22
ARG function, 7-7
ARGFR function, 7-7
ARGS function, 7-7
ARGUMENT command
placement of, 7-7
use of, 7-7
using multiple, 7-8
arguments
in programs, 7-7
in user-defined functions, 7-12
passing as text, 7-9
using ampersand substitution with, 7-9
arithmetic expressions. *See* arithmetic operators, numeric expressions
arithmetic operators, 4-16
assignment operator. *See* = command
assignment statement. *See* = command
AUTOGO programs, 2-11
AW command, 2-5
ATTACH keyword, 2-3
CREATE keyword, 2-3
DETACH keyword, 2-5
LIST keyword, 2-2
NAME keyword, 2-2
WAIT keyword, 2-5
AW function, 2-15
AWDESCRIBE program, 2-15

B

backslash (escape sequence), 3-6
backspace (escape sequence), 3-6
BADLINE option, 7-30
base model, 8-4
batch window for aggregation, 12-2
Boolean
constants, 3-7, 4-21
data type, 3-7, 4-21

Boolean expressions
creating, 4-22
defined, 4-21
example of, 4-23
involving NA values, 4-24
operators, 4-21
values, 4-21
with more than one dimension, 6-6
Boolean operators
evaluation order, 4-21
table of, 4-21
branching in programs, 7-17
BTREE indexes in aggregation, 12-4, 12-6

C

CACHE command
definition, 12-9
purpose of, 12-9
calculation on-the-fly
a typical strategy, 12-25
requirements for, 12-22
calculations
controlling errors during, 4-19
in models, 8-6
CALL command, 7-2
carriage return (escape sequence), 3-6
CDA command, 2-14, 7-17, 11-4
cells, empty, 3-18
characters
representing as decimals, 3-6
representing as hexadecimals, 3-6
representing as Unicode, 3-6
CHGDFN command
aggregation, 12-6
for variables, 3-28
CHILDLOCK command, 9-6
CLEANUP statement (SQL), 10-14
CLOSE statement (SQL), 10-13, 10-14
comments in programs, 7-4
COMMIT command, 2-9
comparison operators, 4-21
COMPILE command
example of, 7-28
in models, 8-5, 8-7
introduction to, 7-28
composites
assigning names to unnamed, 3-20
defined, 3-18
defining single-dimension, 3-22
in expressions, 4-13
limiting base dimensions, 6-18
limiting dimensions used by, 4-14, 6-18
maintaining, 5-9
named, 3-18
naming, 3-20
renaming, 3-20
single-dimension, 3-22
unnamed, 3-18, 3-21
unnaming, 3-20
using commands with, 4-14
concat dimensions, 3-8
defined, 3-25
defining variables for, 3-26
example of, 3-26
limiting, 6-20
maintaining, 5-10
self-relations for, 3-26
conditional expressions, 4-29, 4-30
conditional operators
defined, 4-29
example of, 4-30
conjoint dimensions
deleting values from, 5-8
limiting, 6-19
maintaining, 5-9
maintaining when reading files, 11-10
merging values into, 5-6
CONSIDER command, 3-27
CONTEXT
command, 7-22
function, 7-22
control structures in programs, 7-14
controlled sparsity, 3-18
CONVERT function, 4-3
COPY operator for allocation, 9-10
current analytic workspace, defined, 2-2
current outfile, 7-18
current status, 6-2

cursors (SQL)
    closing, 10-13, 10-14
    declaring, 10-5
    opening, 10-8

D

data aggregation
    best practice, 12-24
    creating the aggregation map, 12-7
    for multiple variables, 12-11
data types
    converting, 4-3, 4-17
    date, 3-7
    numeric, 3-4
    of expressions, 4-2
    of numeric expressions, 4-15, 4-17
    of user-defined function, 7-12
    text, 3-5
data values
    accessing variable, 4-13
    converting when reading files, 11-10
    numeric, 4-15
    saving calculations, 5-12
DATE data type, 3-7
dates
    comparing with times, 4-27
    in arithmetic expressions, 4-18
    in text expressions, 4-20
    reading from files, 11-15
DATETIME data type, 3-7, 4-20
DBGOUTFILE command, 7-31, 8-11
DEADLOCK command, 9-6
debugging programs, 7-29
DECIMAL data type, 3-4, 4-25
decimal data types, comparing, 4-25
DECIMALOVERFLOW option, 4-20
DECIMALS option, 4-24, 4-25
DECLARE CURSOR statement (SQL), 10-5
default outfile, 7-18
DEFINE command, 3-2
    AGGMAP, 12-7
    COMPOSITE keyword, 3-18, 3-19
    DIMENSION keyword, 3-22, 3-25
    MODEL keyword, 8-5
    PROGRAM keyword, 7-3
    RELATION keyword, 3-13
    SPARSE keyword, 3-18
    SURROGATE keyword, 3-11
    VALUESET keyword, 6-22
    VARIABLE keyword, 3-18
definitions
    changing, 3-27
    displaying, 2-15, 2-16
    distinct from data, 3-2
DELETE keyword, 2-5
DESCRIBE command, 2-16
DIMENSION command, 8-5, 8-6, 9-6
dimension order in models, 8-6
dimension status, 6-2
    effect of MAINTAIN command on, 5-4
    effect on expressions, 4-6
    examining, 6-25
    if dimension is empty, 6-21
    if valueset is empty, 6-21
    null, 6-21
    of concat dimension, 6-20
    of conjoint dimension, 6-19
    of dimensions used by composites, 4-14, 6-18
    restoring, 6-4, 6-22, 7-20
    retrieving current values, 6-25
    retrieving default values, 6-25
    saving, 6-22
    saving current status, 6-4, 7-20
    setting to a list of values, 6-4
    setting to a literal value, 6-5
    setting to null, 6-21
    setting using position in dimension, 6-12, 6-13
    when reading files, 11-20
dimension surrogates
    defining, 3-11
    differences from dimensions, 3-12
    in expressions, 4-12
dimension values
    comparing, 4-26
    translating when reading files, 11-11
dimension-based equations, 8-2

dimensions
adding values to, 5-5
assigning values to, 5-14
comparing values, 4-26
concat, 3-8, 3-25
defined, 3-8
defining, 3-22, 3-23, 3-25
defining in a program, 7-29
deleting values from, 5-7
examining values in status, 6-25
hierarchical, 3-22, 3-23
how data is stored, 3-10
in expressions, 4-12
level of detail, 3-9
limiting to a percentage of values, 6-9
limiting to Boolean expressions, 6-5
limiting to bottom performers, 6-10
limiting to related dimension, 6-11
limiting to single value, 4-6
limiting to top performers, 6-10
limiting when reading files, 11-20
limiting, based on position, 6-12, 6-13
limiting, using a valueset, 6-23
limiting, using hierarchical relationship, 6-13, 6-16
looping over values of, 7-15, 7-16
maintaining when reading files, 11-7
merging values into, 5-5
numeric value of text dimension, 4-18
of expression, 4-5, 4-6
of relations, 3-13
position of values in valueset, 6-25
QDR with, 4-6, 4-10
relations between, 3-15
repositioning values in, 5-8, 5-9
restoring previous values, 7-20
retrieving default status list, 6-25
retrieving list of objects related to, 2-16
running programs when limiting, 6-16
saving current values, 7-20
sorting values in, 5-8
storage of, 3-10
surrogate for, 3-11
types of, 3-8
ways to define, 3-22, 3-25
direct allocation, 9-8, 9-13
directory alias, 2-14, 7-17, 11-4
DIVIDEBYZERO option, 4-19
DML
and SQL, 1-3
and the OLAP API, 1-3
definition, 1-2
using, 1-3
double quotes (escape sequence), 3-6

E

ECHOPROMPT option, 7-19, 7-31
EIF file, 2-14
embedded totals dimension, 3-24, 3-26
empty cells, 3-18
EQ command, 3-28
EQ operator, 4-21, 4-22
equations
cyclic dependence (in models), 8-9
dimension-based, 8-2
in models, 8-6
simple blocks, 8-8
step blocks, 8-8
error log, 9-5, 9-17
error messages
creating your own, 7-25
deferring, 7-23
routing to a file, 7-19, 9-17
suppressing, 7-24
system, 7-24
error names, 7-24
ERRORLOG command, 9-6
ERRORMASK command, 9-6
ERRORNAME option, 7-23, 7-24
errors
controlling during calculations, 4-19
handling, 7-23
handling in nested programs, 7-26, 7-27
identifying, 7-24
names of, 7-24
signaling, 7-25, 7-26, 7-27
when comparing numeric data, 4-24, 4-25
ERRORTEXT option, 7-23
escape sequences, 3-6

EVEN operator for allocation, 9-2
EXECUTE statement (SQL), 10-28
EXPORT command, 2-14
expressions
  ampersand substitution, 4-30, 4-31
  Boolean, 4-21, 4-29, 4-30, 6-5, 6-6
  changing the default behavior, 5-12
  conditional, 4-29, 4-30
  data type of, 4-2
  dates in, 4-18
  defined, 4-2
  dimension surrogates in, 4-12
  dimensions in, 4-12
  dimensions of, 4-5, 4-6
  evaluating, default behavior, 5-12
  formulas in, 4-12
  functions in, 4-12
  mixing numeric data types, 4-17
  numeric, 4-15
  objects in, 4-12
  relations in, 4-12, 4-15
  substitution, 4-30, 4-31
  text, 4-20
  using composites in, 4-13
  using text dimension in numeric
    expression, 4-18
  valuesets in, 4-12
  variables in, 4-12

F

fastest-varying dimension, 3-17
FETCH statement (SQL), 10-9
file identifier, 11-4
file names, 11-4
FILENEXT function, 11-15
FILEOPEN function, 11-4
FILEREAD command, 5-3
files
  appending output, 7-18
  fileunit, 11-4
  maintaining dimensions from, 11-7, 11-10
  modifying data from, 11-14
  names and identifiers, 11-4
  reading, 11-1
  reading coded dimension values, 11-12
  reading in programs, 11-5
  reading individual records, 11-15
  reading structured PRN, 11-6
  reading with FILENEXT function, 11-15
  saving output in, 7-17, 7-18
  scaling input data from, 11-13
fileunit, 11-4
FILEVIEW command, 11-15
financial analysis, scenario modeling, 8-12
floating point numbers, comparing, 4-25
floating-point format
  limitations when calculating, 4-18
  use of, 4-18
FOR command
  example of, 7-16
  looping over dimension values, 7-15, 7-16
forecasting data, 5-15
form feed (escape sequence), 3-6
formulas in expressions, 4-12
functions
  defined, 4-15
  in expressions, 4-12
  numeric, 5-16
  user-defined, 7-3, 7-11, 7-12
  writing, 7-11

G

GE operator, 4-21, 4-22
globalization, 2-6
GT operator, 4-21, 4-22

H

HASH indexes in aggregation, 12-6
HEVEN operator for allocation, 9-8
HFIRST operator for allocation, 9-13
hierarchical dimensions
  defined, 3-22
  defining variables for, 3-24
  drilling down, 6-16
  example of, 3-23, 3-24
  limiting based on relationship within, 6-13, 6-16
  self-relations for, 3-24

HLAST operator for allocation, 9-13
horizontal tab (escape sequence), 3-6
host variables (SQL)
input, 10-6
output, 10-9

I

ID data type, 3-5
IFNONE keyword, 7-17
implicit relations, 3-13
IMPORT command, 2-14, 5-3
IMPORT statement (SQL), 10-9
IN operator, 4-21, 4-22
INCLUDE command, 8-4, 8-5, 8-6
INFO function
determining dimensionality with, 4-5
DIMENSION keyword, 4-6
with models, 8-11
input host variables (SQL), 10-6
INSTAT function, 6-3, 6-25
INTEGER data type, 3-4

L

labels
in programs, 7-24
with IFNONE, 7-17
LAG function, 4-18, 8-10
LD command, 3-28
LE operator, 4-21, 4-22
LEAD function, 4-18, 8-10
level relation, defined, 12-4
LIKE operator, 4-21, 4-22, 4-28
LIMIT command
DESCENDANT keyword, 6-14
examples of, 6-5, 6-9, 6-10, 6-11, 6-16, 6-23
HIERARCHY keyword, 6-13, 6-14
NOCONVERT keyword, 6-13
NULL keyword, 6-21
overview, 6-4
POSLIST keyword, 6-12
relation dimension, 6-11
RUN keyword, 6-16
with Boolean expression, 6-5, 6-6
with concat dimension, 6-20
with conjoint dimension, 6-19
with variables with composite, 4-14, 6-18
linefeed (escape sequence), 3-6
LISTNAMES program, 2-16
literals
numeric, 3-4
text, 4-20
local variables, 7-5
localization, 2-6
locking values in an allocation, 9-12, 9-15
logical operators, 4-21
LONGINTEGER data type, 3-4
LT operator, 4-21, 4-22

M

MAINTAIN command
adding values using, 5-5
deleting values using, 5-7, 5-8
effect on dimension status, 5-4
introduced, 5-3
merging values using, 5-5, 5-6
overview of, 5-3
repositioning values using, 5-8
when objects are updated, 5-4
with composites, 5-9
with concat dimensions, 5-10
with conjoint dimensions, 5-9
MAX operator for allocation, 9-8
MODEL command, 3-28, 8-5
MODEL.COMPRPT program, 8-11
MODEL.DEPRPT program, 8-11
models
base, 8-4
basic commands, 8-5
compiling, 8-3, 8-7
creating, 8-2
creating a nested hierarchy, 8-4
debugging, 8-11
defined, 8-2
editing, 8-2
parent, 8-4
running, 8-3, 8-9
scenario, 8-12

solution variables, 8-2
types of solution blocks, 8-8
MODEL.XEQRPT program, 8-11
MODTRACE option, 8-11
multidimensional data model, 3-16
multiple analytic workspaces, 2-6

N

NA values, 3-18
comparing, 4-24
controlling how treated, 4-32
defined, 4-32
in aggregation functions, 4-33
in an allocation, 9-6, 9-7
in arithmetic operations, 4-34
in Boolean expression, 4-24
substituting another value for, 4-34
times when relevant, 4-32
NAFILL function, 4-32, 4-34
NAME dimension, 2-16
named composites, defined, 3-18
NASKIP option, 4-32, 4-33
NASKIP2 option, 4-32, 4-34
NASPELL option, 7-7
NE operator, 4-21, 4-22
NLS options, 2-6
NOL_SORT option, 4-27
NOPRINT keyword (TRAP), 7-24, 7-27
NOSPELL option, 3-7
NOT operator, 4-21, 4-22
NTEXT data type, 3-5
NUMBER data type, 3-5
NUMBER dimension, surrogate for, 3-11
numeric data types
automatic conversion of, 4-17
comparing, 4-24, 4-25
list of, 3-4
mixing, 4-17
numeric expressions
data type of the result, 4-15, 4-17
dates in, 4-18
defined, 4-15
evaluating, 4-16
mixing data types in, 4-17
NA values in, 4-34

O

OBJ function
PROPERTY keyword, 12-5
workspace object information, 2-17
objects
assigning values to, 4-3, 5-10
changing definition of, 3-27
definitions, 3-2
displaying definitions of, 2-16
in expressions, 4-12
list of, 3-3
maintaining, 5-4
retrieving information about, 2-17
retrieving list of, 2-16
updating, 5-4
OKNULLSTATUS option, 6-21, 7-17
OLAP Worksheet, 1-6
OPEN statement (SQL), 10-8
operators
arithmetic, 4-16
Boolean, 4-21
comparison, 4-21
conditional, 4-29, 4-30
for allocation, 9-7
logical, 4-21
substitution, 4-30, 4-31
options
restoring previous values, 7-20
saving current values, 7-20
OR operator, 4-21, 4-22
OUTFILE command, 7-17, 7-18
OUTFILEUNIT option, 12-21
output
host variables, 10-9
saving in a file, 7-17, 7-18

P

parent model, defined, 8-4
parent relation, defined, 12-4
PARSE command, 4-5, 4-6
pattern matching, 4-28

permission programs, 2-12
PERMIT command, 2-13, 3-28
PERMIT_READ program, 2-12, 2-13
PERMIT_WRITE program, 2-12, 2-13
POP command, 7-20, 7-21
POPLEVEL command
  nesting, 7-22
  using, 7-21
populating analytic workspaces, 5-1
POUTFILEUNIT option, 9-4, 12-4, 12-21
PREPARE statement (SQL), 10-28
PRGERR keyword (SIGNAL), 7-26
PRGTRACE option, 7-31
PRN files, reading, 11-6
PROGRAM command, 3-28
programs
  analytic workspace permission, 2-13
  arguments, 7-7
  AUTOGO, 2-11
  automatic running of, 2-11, 6-16
  branching in, 7-17
  branching labels, 7-14
  comment lines in, 7-4
  compiling, 7-9, 7-28
  control structures, 7-14
  debugging, 7-29
  declaring arguments in, 7-7, 7-8
  defined, 7-2
  defining, 7-3
  designing, 7-5, 7-14
  errors in, 7-23
  executing, 7-2
  LISTNAMES, 2-16
  permission, 2-12
  PERMIT_READ, 2-12, 2-13
  PERMIT_WRITE, 2-12, 2-13
  preserving environment, 7-19
  restoring previous values, 7-20
  running, 7-29
  sample, 7-12
  saving compiled code, 7-28
  saving current values, 7-20
  testing by running, 7-29
  variables in, 7-4, 7-5
PROPERTY command, 3-28
PROPORTIONAL operator for allocation, 9-15
protecting values in an allocation, 9-12, 9-15
PUSH command, 7-21
  placement, 7-23
  using, 7-20
PUSHLEVEL command
  nesting, 7-22
  placement, 7-23

Q

QON, 2-6
QUAL function, 4-10
qualified data references
  ampersand substitution, 4-10
  creating, 4-6
  defined, 4-6
  qualifying a relation, 4-9
  replacing dimension of variable, 4-7, 4-8
  using with = command, 4-8, 5-14
  using with relation, 4-9
  with dimensions, 4-6
  with relations, 4-9
  with variables, 4-7, 4-8
qualified object names, 2-6
quotation marks (escape sequence), 3-6

R

random sparsity, 3-18
RAW DATE attribute
  when reading files, 11-15
records, reading, 11-15
recursive allocation, 9-10, 9-15
RELATION command, 12-7
  arguments for allocation, 9-7
  for allocation, 9-6
  operators for allocation, 9-7
  syntax for aggregation, 12-12
relational data, 10-1
  See also SQL
  copying into analytic workspace, 10-3 to 10-20
  inserting from analytic
    workspace, 10-28 to 10-32

updating from analytic workspace, 10-28 to 10-32
relations
assigning values to, 5-14
between two dimensions, 3-15
comparing to text literals, 4-29
defined, 3-13
defining, 3-15
dimensionality of, 3-13
example of, 3-15, 3-24, 3-26
how data is stored, 3-14
implicit, 3-13
in aggregation, 12-4
in allocation, 9-5, 9-6, 9-9
in expressions, 4-12, 4-15
limiting to single value, 4-9
QDR with, 4-9
replacing dimension of, 4-9
self, 3-15, 3-24, 3-26
used when reading files, 11-13
REMOPERATOR in an allocation, 9-9
REPORT command
for viewing objects, 2-17
with sparse data, 4-14
RETURN command, 7-11
ROLLBACK, effect on changes, 2-10
ROOTOFNEGATIVE option, 4-20
ROUND function, 4-24, 4-25
run-time aggregation, 12-2

S

scenario model, defined, 8-12
scenarios for financial modeling, 8-12
sessions
preserving environment, 7-19
restoring environment, 7-20
sharing analytic workspaces across, 2-4
SHORTDECIMAL data type, 4-25
SHORTINTEGER data type, 3-4
SIGNAL command, 7-25
simple blocks (in models), 8-8
simultaneous equations in models, 8-10
single quotes (escape sequence), 3-6
slowest-varying dimension, 3-17
solution variables
defined, 8-2
example of, 8-12
source object for allocation, 9-5
SOURCEVAL command, 9-6
sparse data, 3-18
controlled sparsity, 3-18
defined, 3-18, 4-32
eliminating, 3-18 to 3-21
random sparsity, 3-18
setting dimension status, 6-18
SQL, 10-1
See also relational data
error handling, 10-34
OLAP DML command, introduced, 5-3
precompiling code, 10-28
stored procedures, 10-32
triggers, 10-32
SQL statements
issuing through OLAP DML, 10-2 to 10-33
STATFIRST function, 6-3, 6-25
STATLAST function, 6-3, 6-25
status. *See* dimension status
step blocks (in models), 8-8
storage
of dimensions, 3-10
of relations, 3-14
of variables, 3-17
stored procedures, 7-2
structured files, reading, 11-6
substitution expressions, 4-30, 4-31
substitution operator, 4-30, 4-31
surrogates. *See* dimension surrogates
SYSINFO function, 2-13

T

tab (escape sequence), 3-6
temporary variables, 7-5, 11-20
text
comparing values, 4-27, 4-28
comparing values to a pattern, 4-28
data types, 3-5
NLS_SORT option in comparisons, 4-27
passing arguments as, 7-9

TEXT data type, 3-5
text expressions
  dates in, 4-20
  defined, 4-20
text literals
  comparing to relations, 4-29
  defined, 4-20
TRAP command, 7-23, 7-26, 7-27

U

unnamed composites, 3-18, 3-21
  defining, 3-21
  example of, 3-21
  naming, 3-20
UPDATE command, 2-9
user-defined functions, 7-11
  arguments in, 7-12
  data type of, 7-12
  defined, 7-2
  executing, 7-3

V

VALUE keyword
  used in reading files, 11-12
  used when reading files, 11-14
values
  assigning to dimensions, 5-14
  assigning to objects, 5-10
  assigning to relations, 5-14
  assigning to variables, 5-12
  assigning to variables with composites, 5-12, 5-13
  assigning, using a QDR, 5-14
  in current status list, 6-25
  in default status list, 6-25
  NA, 3-18
  restoring previous, 7-20
  saving current, 7-20
VALUES function, 6-3, 6-25
valuesets
  creating, 6-22
  defined, 6-22
  defining, 6-22
  in expressions, 4-12
  limiting using, 6-23
  listing dimension positions in, 6-25
VARIABLE command, 7-6
variables
  accessing, 4-13
  aggregating data for multiple, 12-11
  assigning values to, 5-11, 5-12, 5-13, 5-14
  controlling sparsity in, 3-18
  defined, 3-16
  defining in a program, 7-29
  defining with composite, 3-18 to 3-21
  defining with unnamed composite, 3-21
  dimensioned, 3-17
  how data is stored, 3-17
  in expressions, 4-12
  limiting to single value, 4-7, 4-8
  local, 7-5
  NA values in, 3-18
  persistence of, 7-4, 7-5
  QDR with, 4-7, 4-8
  replacing dimension of, 4-7, 4-8
  sparse data in, 4-14
  storage of, 3-17
  temporary, 7-5
  undimensioned, 3-17
  with embedded totals, 3-24, 3-26
  with NA values, 3-18
  with single-dimension composite, 3-22

W

WHERE clauses (SQL), 10-7
wildcards, 4-28

Y

YESSPELL option, 3-7

Z

zero, dividing by, 4-19