Oracle9*i* OLAP Services

Developer's Guide to the OLAP DML

Release 1 (9.0.1)

June 2001

Part No. A86720-01

ORACLE

Oracle9i OLAP Services Developer's Guide to the OLAP DML, Release 1 (9.0.1)

Part No. A86720-01

Contents

3 Defining Data Objects

4 Working with Expressions

5 Populating OLAP DML Data Objects

6 Limiting an Application's View of the Data

7 Working with Models

8 Designing Programs

9 Debugging Programs

10 Using Embedded SQL

11 Reading Data from Files

12 Writing Reports

A Creating and Using Analytic Workspace Metadata

Glossary

Index

Send Us Your Comments

Oracle9i OLAP Services Developer's Guide to the OLAP DML, Release 1 (9.0.1)

Part No. A86720-01

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

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

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

- FAX - 781-684-5880. Attn: Oracle OLAP Services
- Postal service:
  Oracle Corporation
  OLAP Services Documentation Manager
  200 Fifth Avenue
  Waltham, MA 02451-8720
  USA

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

What this manual is about

The *Oracle9i OLAP Services Developer's Guide to the OLAP DML* provides an overview of the programming environment for the OLAP DML, describes the OLAP DML data objects, and explains how to use the key commands in the OLAP DML. It also describes how to write and debug OLAP DML programs and illustrates programming strategies for accessing and working with data.

Intended audience

This guide is intended for users who want to perform the following tasks:

- Write OLAP DML programs
- Manage analytic workspaces with the OLAP DML
- Access data and perform analysis using the OLAP DML

Structure of this document

The *Oracle9i OLAP Services Developer's Guide to the OLAP DML* is structured as follows:

- Chapter 1 provides an overview of the OLAP DML environment.
- Chapter 2 describes how to create, attach, and manage analytic workspaces, how to save analytic workspace changes, how to share analytic workspaces, and how to obtain information about an analytic workspace.
- Chapter 3 describes how to define OLAP DML objects.

- Chapter 4 describes the OLAP DML data types and operators and how to create expressions in the OLAP DML.
- Chapter 5 provides an overview of how to populate OLAP DML data objects and how to calculate values.
- Chapter 6 describes how to limit an application's view of the data.
- Chapter 7 describes how to write models using the OLAP DML.
- Chapter 8 describes how to write OLAP DML programs.
- Chapter 9 describes how to use the OLAP DML debugger.
- Chapter 10 describes how to access relational data by using SQL commands in the OLAP DML.
- Chapter 11 describes how to use the OLAP DML to read data from files.
- Chapter 12 describes how to use the OLAP DML to write reports.
- Appendix A describes how to create and use analytic workspace metadata.
- The Glossary provides definitions of OLAP terminology.

Related Documentation

You will find the following documentation helpful when using the Oracle OLAP API and Oracle OLAP Services:

- *Oracle9i OLAP Services Concepts and Administration Guide* — Describes how to use OLAP Services. It introduces the basic concepts underlying business analysis and multidimensional querying, as well as the basic tools used for application development and system administration.
- *Oracle9i OLAP Services Developer's Guide to the Oracle OLAP API* — Introduces Java programmers to the Oracle OLAP API, the application programming interface for Oracle OLAP Services. Through OLAP Services, the OLAP API provides access to data stored in an Oracle database. The OLAP API's capabilities for querying, manipulating, and presenting data are particularly suited to applications that perform online analytical processing.
- *Oracle9i OLAP Services OLAP API Reference* — Provides online reference documentation for the Oracle OLAP API, the Java application programming interface for Oracle OLAP Services.
- *Oracle9i Data Warehousing Guide* — Discusses the database structures, concepts, and issues involved in creating a data warehouse to support OLAP solutions.

Conventions

Text conventions

You will find the following text conventions in this document.

Convention	Usage
Boldface text	Indicates menu items, command buttons, options, field names, and hyperlinks. Bold text is also used for notes and other secondary information in tables (for example, **Result**).
`Fixed-width` text	Indicates folder names, file names, operating system commands, and URLs. Also indicates examples and anything that you must type exactly as it appears. **For example:** If you are asked to type `show eversion`, then you would type all the characters exactly as shown in the fixed-width font.
Italic text	Indicates variables, including variable text that is used in the following ways: ■ In the syntax of OLAP DML commands to indicate arguments or parameters. ■ When dialog boxes or their components are unlabeled or have labels that change dynamically based on their current context. The wording of variable text does not exactly match what you see on your screen. Italic type is also used for emphasis, for new terms, and for titles of documents.
Underlined text	Indicates a default value in descriptions of OLAP DML syntax.
UPPERCASE text	Indicates Express commands and objects and acronyms.

Mouse usage

Always use the left mouse button unless you are specifically instructed to use the right mouse button.

The term “left mouse button” refers to the dominant button. If you have reconfigured your mouse to reverse the functions of the left and right buttons, then you will need to use the reverse button when you follow the procedures in this manual.

Formats for key combinations and sequences

Key combinations and key sequences appear in the following formats.

IF you see the format . . .	THEN . . .
Key1+Key2,	press and hold down the first key while you press the second key. **For example:** "Press Alt+Tab" means to press and hold down the Alt key while you press the Tab key.
Key1, Key2,	press and release the keys one after the other. **For example:** "Press Alt, F, O" means to press and release the Alt key, press and release the F key, then press and release the O key.

Documentation Accessibility

Oracle's goal is to make our products, services, and supporting documentation accessible to the disabled community with good usability. 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 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/`

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.

1

Basic Concepts

Chapter summary

This chapter provides an overview of the basic concepts that you should understand before you use the OLAP DML. The checklist at the end of this chapter explains how to use this guide to learn the specific tasks that this chapter discusses.

List of topics

This chapter includes the following topics:

- What Is the OLAP DML?
- Using the OLAP DML
- How Do I Use the OLAP DML to Analyze Data?
- Where Do I Go From Here?

What Is the OLAP DML?

Definition: 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.

If you are familiar with Oracle Express Server, think of the OLAP DML as being the same as the Express Server stored procedure language (SPL). In fact, the OLAP DML is nearly 100 percent compatible with Oracle Express Server's stored procedure language.

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. The complete syntax for each command is provided in the OLAP DML Reference, which is a help system that you can access from OLAP Worksheet.

The purpose of the OLAP DML

The purpose of the OLAP DML is to allow application developers to extend Oracle OLAP Services' Java OLAP API.

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

- OLAP Server data sources
- Analytic workspaces
- The relationship of the Oracle OLAP API to the OLAP DML

OLAP Services data sources

OLAP Services can access two different data sources — the Oracle relational database and analytic workspaces. For most applications the Oracle relational database will be the primary (and often the only) data source. When accessing data from the Oracle relational database, OLAP Services generates SQL to access data stored in tables. The OLAP API provides a wide variety of analytic functions that allow the application to derive calculated measures when using the Oracle database as the data source.

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 this case, you can use the OLAP DML to calculate this data. The OLAP DML does not operate directly on data in relational tables. Instead, it operates on data within an analytic workspace.

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, a separate data file is created that is not part of the relational database.

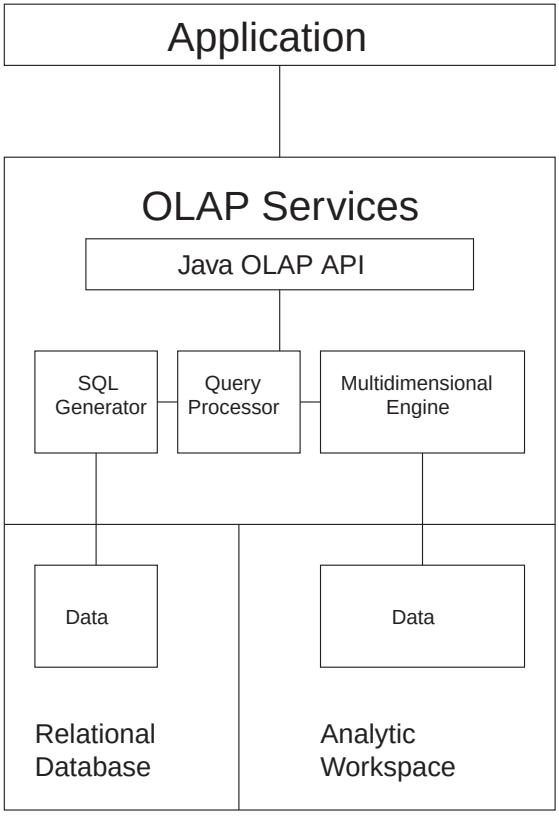
OLAP API and OLAP DML

The OLAP API performs three primary functions:

- It provides access to metadata the describes the multidimensional data model and data sources.
- It fetches data from data sources.
- It provides the means of performing complex analytic calculations.

When accessing data in the relational database, OLAP Services processes OLAP API queries by generating SQL. When accessing data in an analytic workspace, OLAP Services processes OLAP API queries by using OLAP Services' multidimensional engine and the OLAP DML.

The following illustration provides an architectural overview of OLAP Services, showing both the relational database and analytic workspaces as data sources.



Using the OLAP DML

When should I use the OLAP DML?

There are three situations where 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 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 to the data warehouse.
- You want to use a data file that was created with Oracle Express Server with OLAP Services.

The common calculations for which you will need OLAP DML

The most common types of calculations that the OLAP DMS is used for includes:

- Forecasts
- Models (a group of calculations in which the results of one calculation is required by another calculation — refer to Chapter 7 for more information)
- Allocations
- 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 using the OLAP API.

Because analytic workspaces are not stored in the relational database, it is possible for you to commit data to the analytic workspace without committing it to the data warehouse. 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 that user to commit the forecast to the data warehouse.

How Do I Use the OLAP DML to Analyze Data?

Procedure: Using the OLAP DML

To use the OLAP DML, you:

1. Create an analytic workspace (see "Creating an analytic workspace" on page 1-6).
2. Define data sources within the analytic workspace (see "Defining data sources within the analytic workspace" on page 1-7).
3. Load data into the analytic workspace (see "Loading data into analytic workspaces" on page 1-7).
4. Define and execute the OLAP DML commands and programs (see "Executing OLAP DML commands and functions" on page 1-7).

What can I do with the data after I have analyzed it?

After you have used the OLAP DML to analyze data from a table, you can then:

- View data in an analytic workspace using the OLAP API
- Can commit data to the data warehouse

Creating an analytic workspace

Creating an analytic workspace is a very simple process and is accomplished using a command in the OLAP DML. An example of this command follows:

```
DATABASE CREATE \DATA\SALESFORECAST
```

The above command will create a new and empty analytic workspace named SALESFORECAST in the \DATA directory.

This would be similar to creating a new tablespace and data file in the relational database. That is, the physical file is created, but there are no objects stored in the file yet.

For more information about creating an analytic workspace, refer to Chapter 2.

Defining data sources within the analytic workspace

Within an analytic workspace, you create workspace objects and OLAP API data sources. Examples of workspace objects include variables, formulas, and dimensions. These are the basic building blocks within an analytic workspace.

These workspace objects can be augmented with analytic workspace metadata to expose data to the OLAP API.

For more information about creating data sources, refer to Chapter 3. To learn how to create and use analytic workspace metadata, refer to Appendix A.

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 by fetching it from the relational database or by loading it using OLAP Services' file reader. In most cases, the relational database will be the data source.

Data is loaded into the analytic workspace using commands in the OLAP DML. There are commands for fetching data from relational tables and commands for reading flat files. When loading data from the relational database, the data can be from any table (that is, it does not need to be part of a data warehouse).

For more information about loading data into an analytic workspace, refer to Chapter 10.

Executing OLAP DML commands and functions

The OLAP DML consists of various commands and functions. Commands create, delete, and modify objects, call programs, fetch and load data, and perform other needed tasks. Functions typically manipulate data (for example, return the TOTAL of SALES).

You can call individual functions and you can define and execute formulas and programs. Whether you use functions, formulas, or programs will depend on what you are trying to accomplish.

OLAP Services applications call or executes commands and functions using the SPL EXECUTE method in the OLAP API. The SPL EXECUTE method allows the OLAP API to pass OLAP DML commands to the multidimensional engine that processes the OLAP DML.

The OLAP Worksheet application allows you to work interactively with the OLAP DML (much the same way that SQL Plus allows you to interact with the relational

database by typing SQL statements). Using the OLAP Worksheet, you can execute most OLAP DML commands. You can also define workspace objects, and edit programs and formulas.

You might, for example, use the OLAP Worksheet to define a OLAP DML program that loads data from the relational database and forecasts sales. Your application would then call execute the OLAP DML program through the OLAP API using the SPL EXECUTE method.

For more information about using OLAP DML commands, functions, and programs, refer to Chapter 4, Chapter 5, Chapter 6, and Chapter 7.

Viewing data in an analytic workspace

Any data in an analytic workspace that has been exposed as an OLAP API data source with analytic workspace metadata will automatically become available through the OLAP API. To the OLAP API client, there is no difference between a relational database data source and an analytic workspace data source. The OLAP API reveals all data sources to the application without requiring the application to understand the data's physical storage.

Temporary vs. persistent analytic workspaces

Analytic workspaces may 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 does 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 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.

Committing data to the data warehouse

Data created within an analytic workspace can be committed to the relational database using the OLAP DML. Data can be committed to any table, regardless of whether that table is part of a data warehouse.

The are two primary reasons why you might want to commit data in an analytic workspace to the relational database:

- You want to use the relational database for long-term persistent storage. This is advisable because the relational database has more robust disaster recovery features as compared to analytic workspaces.
- You want to make the results of calculations made within an analytic workspace available to SQL-based applications (that is, applications that use SQL rather than the OLAP API).

Two common examples of situations where you might want to commit data to the relational database include:

- Sharing the results of a final forecast with users of SQL-based applications.
- Managing the long persistence of data resulting from specialized aggregations in the relational database.

Where Do I Go From Here?

The OLAP DML checklist

To learn how to perform all the tasks described in this chapter, use the following checklist:

- ❑ Read Chapter 2 to learn how to create an analytic workspace.
- ❑ Read Chapter 3 to learn how to populate an analytic workspace with the objects that will contain data from the Oracle relational database.
- ❑ Read Chapter 10 to learn how to select data from an Oracle relational database and move that data into the objects in a DML work file.

- ❑ Read Chapter 4, Chapter 5, Chapter 6, and Chapter 7 to learn how to use the DML commands and functions to analyze data.
- ❑ Read Chapter 10 to learn how to move the analyzed data from the work file into the Oracle relational database.

For more information

The following publications provide more information about the Oracle OLAP API and the OLAP DML:

- *Oracle9i OLAP Services Developer's Guide to the Oracle OLAP API*
- The OLAP Worksheet's Help system contains an on-line reference for the DML language

2

Defining and Working with Analytic Workspaces

Chapter summary

This chapter discusses creating, attaching, and managing analytic workspaces.

List of topics

This chapter includes the following topics:

- Defining an Analytic Workspace
- How to Gain Access to an Analytic Workspace
- Gaining Access to a Workspace from a Java Application
- Gaining Access to a Workspace from OLAP Worksheet
- Using the OLAP DML to Work with Analytic Workspaces
- Saving Analytic Workspace Changes
- Minimizing Analytic Workspace Growth
- Sharing Analytic Workspaces
- Working with AUTOGO Programs
- Adding Security to an Analytic Workspace
- Obtaining Analytic Workspace Information

Defining an Analytic Workspace

How to define an analytic workspace

Analytic workspaces are defined using commands in the OLAP DML. There are two methods by which this can be accomplished:

- Use the OLAP API's `SPLExecutor` method to issue OLAP DML commands. This allows applications using the OLAP API to create new analytic workspaces and alter existing workspaces. When workspaces are defined through the `SPLExecutor` method, they can be temporary (that is, for the life of the session) or they may be persisted.
- Use OLAP Worksheet to issue OLAP DML commands. OLAP Worksheet connects to an OLAP Services instance as a new session and allows you to work interactively with the OLAP DML; this is similar to issuing SQL commands from within SQL Worksheet. When an analytic workspace is created using OLAP Worksheet, it must be persisted so that OLAP Services applications using the OLAP API will be able access it.

This guide discusses how to use OLAP DML commands to define an analytic workspace.

Examples: Defining an analytic workspace

The following example creates a new analytic workspace named `shoes`; the full name of the new analytic workspace is `shoes.db`.

```
database create shoes
```

The following example creates the `shoes.db` analytic workspace in a directory named `apps` on the `i` drive of an NT system.

```
database create 'i:/apps/shoes'
```

For the complete syntax for the DATABASE command, see the OLAP DML Reference.

About the term "database"

Throughout this guide, you will notice that the OLAP DML command 'database' is used to create and manage analytic workspaces. When referring to the OLAP DML, you can think of the terms 'database' and 'analytic workspace' as being equivalent. The 'database' command is used in the OLAP DML to allow for compatibility with

the Express Server stored procedure language. (Express Server was the predecessor to OLAP Services.)

Do not confuse analytic workspaces with the Oracle relational database. Analytic workspaces are stored in files that are separate from Oracle relational database files.

Managing analytic workspace structure and size

An analytic workspace can be made up of many files. There is always a main analytic workspace file. There can also be one or more extension analytic workspace files. You can use extension files to divide a single analytic workspace among several files, so the analytic workspace can be larger than the space that is available on any single disk. Typically, you need extension files only when the analytic workspace is located on a disk with limited available space or when the analytic workspace will grow to a very large size. An analytic workspace that is stored in more than one file is called a *multifile analytic workspace*.

When you use the DATABASE command with the CREATE keyword, a new analytic workspace file is created. As the analytic workspace is populated, data is added to that file and, optionally, additional analytic workspace extension files are created, if needed. Depending on the options that you specify when you create an analytic workspace, you can change the default characteristics of these files:

- The maximum size of analytic workspace files
- The increment size of analytic workspace files
- The location of the main analytic workspace file
- The location of analytic workspace extension files

  Note: If you want to specify location of analytic workspace extension files only for a given session, then use the DBEXTENDPATH option. If you want to specify the location of analytic workspace extension files only for an instance of OLAP Services, then use the ExtensionFilePath setting of the OLAP Services Instance Manager.

How to Gain Access to an Analytic Workspace

Two alternatives for accessing a workspace

Once an analytic workspace has been defined, it can be accessed in one of the following two ways.

- From OLAP Worksheet, or
- From a Java application

For more information

To learn how to access an analytic workspace from OLAP Worksheet, see "Gaining Access to a Workspace from OLAP Worksheet" on page 2-4.

To learn how to access an analytic workspace from a Java application, see "Gaining Access to a Workspace from a Java Application" on page 2-6.

Gaining Access to a Workspace from OLAP Worksheet

What is OLAP Worksheet?

OLAP Worksheet is a command line interface to OLAP Services that you can use to perform the following tasks:

- Access an analytic workspace
- Execute most OLAP DML commands
- Edit programs
- Debug programs

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

You can enter OLAP DML commands in the Input (query) pane at the bottom of the Command Input window. The results are displayed in the Output (result) pane at the top of the Command Input window.

This chapter describes how to use OLAP Worksheet to access analytic workspaces and execute OLAP DML commands. Refer to Chapter 9 for information about writing, editing, and debugging programs with OLAP Worksheet, as well as how to display its Edit window.

This guide provides basic information about using OLAP Worksheet. For details, refer to the OLAP Worksheet Help system.

Overview of accessing a workspace

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

Establishing a connection

Use the following procedure to establish a connection to OLAP Services:

1. In the OLAP Worksheet menu bar, choose **File.**
2. Choose **Connect.**
3. Enter valid user credentials and information in the Login dialog box that appears.

Opening the workspace

Once you have made a connection to OLAP Services, you can open an analytic workspace by entering a DATABASE ATTACH command in the Command Input window in OLAP Worksheet.

Alternatively, you can define a new analytic workspace.

For example, suppose that you have already defined a workspace named SALES that exists at the top level of the current directory. Enter the following command to open the SALES analytic workspace:

```
database attach sales
```

Suppose you want to define a new workspace named EXPENSE. You would enter the following commands to create the new workspace and then open it:

```
database create expense
database attach expense
```

For more information about opening workspaces and working with more than one workspace at a time, see “Using the OLAP DML to Work with Analytic Workspaces” on page 2-9.

Closing the workspace

When you have finished working with an analytic workspace and have saved changes, you can close it by entering a DATABASE DETACH command in the Command Input window in OLAP Worksheet.

For example, to close an analytic workspace named SALES, enter the following command:

```
database detach sales
```

Closing the connection

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

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

Gaining Access to a Workspace from a Java Application

Overview of accessing a workspace

Typically, a Java application uses the Oracle OLAP API to access relational data. In addition, the Oracle OLAP API supports access to data that resides in an OLAP Services analytic workspace.

Through the OLAP API, a Java application can access workspace data that has been provided with analytic workspace metadata. Because analytic workspace metadata is compatible with the OLAP API multidimensional metadata (MDM) model, a Java application can manipulate workspace data using the OLAP API Java classes. For a description of the MDM model and the OLAP API classes, see the *Oracle9i OLAP Services Developer's Guide to the Oracle OLAP API* and the *Oracle9i OLAP Services OLAP API Reference*.

As an alternative access method, 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 send DML commands directly to OLAP Services for execution in the workspace.

Whichever access method is used, the application establishes a connection, opens the workspace, accesses the data (either through MDM metadata or through

SPLExecutor), closes the workspace, and closes the connection. This topic describes these steps.

Establishing a connection

To make a connection, follow the steps described in the chapter about connecting to a data store in the *Oracle9i OLAP Services Developer's Guide to the Oracle OLAP API*. The chapter describes how the OLAP service makes a connection to its parent Oracle database instance. This connection is a requirement even if your application will only access data in an OLAP Services workspace.

When you have connected to the workspace, you will have a `Connection` object that represents the connection.

Opening the workspace

Use the `openDatabase` method on the `Connection` object to open the workspace that you want to access. The `openDatabase` method requires the following two parameters:

- The name of the workspace. This is the name that was used in the OLAP Services DML to create the workspace. For example, the following DML command creates a workspace named FORECAST.

  ```
  DATABASE CREATE FORECAST
  ```

- A `Properties` object that specifies parameters that are appropriate for opening the workspace. The `Properties` object contains one or more properties, each of which represents one parameter. If you are opening a workspace that has analytic workspace metadata, then you must specify a property called "DatabaseType", with a value of "ECM".

  To discover if there are other required or optional parameters, use the `getConnectionParameterInfo` method on the `Connection` object, as described in the reference page for the `Connection` class in the *Oracle9i OLAP Services OLAP API Reference*.

The `openDatabase` method returns the `Database` object that represents the workspace.

Accessing workspace data using MDM metadata

The OLAP API provides classes that support the MDM model for describing a set of data. These classes are in the `mdm` package of the OLAP API.

Ordinarily, the data that an application analyzes is stored in an Oracle database, and it has been provided with MDM-compatible metadata through the OLAP management feature of Oracle Enterprise Manager. However, if your workspace has been provided with analytic workspace metadata, the OLAP API can use the workspace data for analysis, because its analytic workspace metadata is MDM-compatible.

To access analytic workspace metadata and the data that it represents, create an `MdmMetadataProvider`, as described in the chapter on discovering metadata in the *Oracle9i OLAP Services Developer's Guide to the Oracle OLAP API*. When you create the `MdmMetadataProvider`, specify the `Database` object that represents the workspace.

To navigate the metadata, create queries, and fetch data, use the procedures that are described in the *Oracle9i OLAP Services Developer's Guide to the Oracle OLAP API*.

Accessing workspace data using SPLExecutor

The OLAP API provides the `SPLExecutor` class, through which you can execute DML commands or evaluate DML expressions in a workspace. You create an `SPLExecutor` object, specifying your `Connection` object as a parameter. Then you call the `executeCommand` method on your `SPLExecutor` to send commands to the OLAP service, or you call one of several methods for evaluating expressions.

Depending on the evaluation method that you use, the return value is a `boolean`, `int`, `double`, or other Java data type. An executed command always returns a `String`.

For more information about the `SPLExecutor` methods, see the reference page for this class in the *Oracle9i OLAP Services OLAP API Reference*.

Closing the workspace

When you are finished with your work, call the `close` method on the `Database` object that represents your workspace. This method closes the `Database` object, and the OLAP service detaches the workspace that is associated with it.

Closing the connection

When you no longer need the connection, call the `close` method on the `Connection` object. This method terminates the connection that was made on behalf of your application with the Oracle database instance, and the method terminates the connection between your application and the OLAP service.

Using the OLAP DML to Work with Analytic Workspaces

Definition: Active analytic workspace

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 2-10.

Listing the active analytic workspaces

You can view a list of the active analytic workspaces by using the DATABASE command with the LIST keyword. For the complete syntax for the DATABASE command, see the OLAP DML Reference. The simplified syntax for this command is shown below.

```
database list
```

This command displays a list of the active analytic workspaces, along with their update status and full path name. The `express.db` analytic workspace, which is a system analytic workspace that contains objects used internally, always appears in the analytic workspace list.

The meaning of the update status, CHANGED or UNCHANGED, depends on whether the analytic workspace is attached with read/write or read-only access and whether the analytic workspace is being shared with other users.

Definition: 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 DATABASE command with the LIST keyword. By default, when you define new OLAP DML objects, they reside in the current analytic workspace, unless you specify the name of another active analytic workspace. Additionally, programs such as DBDESCRIBE list only the objects in the current analytic workspace.

Your session does not have to have a current analytic workspace. If you start OLAP Services without specifying an analytic workspace name, then the `express.db` analytic workspace is first on the list. However, the `express.db` analytic workspace is not current; there is no current analytic workspace until you specify one with the DATABASE command. Even though an active analytic workspace is not current, you can still change and update its data, edit and run its programs, and modify its analytic workspace definitions.

Retrieving the name of the current analytic workspace

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

Example: Retrieving the name of the current analytic workspace

Suppose that you have two analytic workspaces attached, one named `programs.db` and another named `demo.db`. The following commands use the DATABASE 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 = database(name)
show mytext
PROGRAMS
```

How to attach an analytic workspace

The system administrator can change OLAP Services configuration settings so that OLAP Services starts up with one or more application analytic workspaces already attached. To reconfigure OLAP Services, use the OLAP Services Instance Manager.

You can also use the DATABASE command to attach and detach analytic workspaces during a session. During your OLAP Services session, you can use the DATABASE command to switch freely between active analytic workspaces.

You can attach an analytic workspace by using the DATABASE command with or without the ATTACH keyword. As shown below, the action that is taken varies depending on whether or not you use the ATTACH keyword:

- If you attach an analytic workspace by using the DATABASE command with the ATTACH keyword, then the analytic workspace that you specify is automatically attached and made to be the current analytic workspace.
- If you attach an analytic workspace by using the DATABASE command with the ATTACH keyword, then different actions are taken, depending on whether or not there is a current analytic workspace:
  - If there is no current analytic workspace, then the new analytic workspace is attached and made current.
  - If there is a current analytic workspace, then it is first detached. The new analytic workspace is attached and made current.

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 DATABASE command as described in "Specifying the analytic workspace attachment mode" on page 2-11.

Note: You can create programs that are automatically executed when you attach an analytic workspace. For more information, see "Programs that run when a user attaches to an analytic workspace" on page 2-12.

Examples: Attaching an analytic workspace

The following example attaches an existing analytic workspace named `finance.db` and makes it the current analytic workspace. If another analytic workspace was current before this command executes, then that analytic workspace remains attached but is no longer current.

```
database attach finance
```

The following example attaches the `finance.db` analytic workspace and makes it current. The analytic workspace that was current is detached before this command was executed.

```
database finance
```

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 RW EXCLUSIVE keywords of the DATABASE 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.

For more information on sharing analytic workspaces across sessions and specifying the analytic workspace attachment mode, see "Sharing Analytic Workspaces" on page 2-17.

Programs that run when a user attaches to an analytic workspace

When a user attaches to an analytic workspace, permission programs and AUTOGO programs will be run automatically if they exist:

1. Any permission programs that are associated with the analytic workspace will be automatically executed.
2. Any AUTOGO program associated with the analytic workspace or any program specified in the DATABASE ATTACH command will be executed.

Permission programs are, as the name suggests, programs that check the permission of the user. AUTOGO programs and programs specified in the DATABASE ATTACH command can contain any type of functionality. For more information on permission programs, see "Adding Security to an Analytic Workspace" on page 2-20. For more information on other programs that run when a user attaches to an analytic workspace, see "Working with AUTOGO Programs" on page 2-19.

Attaching multiple analytic workspaces

You can attach more than one analytic workspace at a time. However, when working with multiple analytic workspaces, keep the following points in mind:

- Oracle Corporation does not recommend that you give the same name to more than one analytic workspace. However, if there is more than one analytic workspace with the same name, then specify the full path name of the analytic workspace you want.
- If you are going to attach more than one analytic workspace, then you must take more 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 in order 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. If you do, then you can create unanticipated interactions between the objects.
- If you have analytic workspace 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 take special care when you edit them, or use them in any other way, to ensure that you access the appropriate version.

Detaching analytic workspaces

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

```
database detach finance
```

Saving Analytic Workspace Changes

When should you save your analytic workspace changes?

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

If you have read / write access to the analytic workspace, then you will get a warning message if you try to switch analytic workspaces, detach a changed analytic workspace, or exit without updating the analytic workspace. 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 by updating it.

Using the UPDATE command

If you have changed an analytic workspace and want to save those changes, then execute the UPDATE command. The UPDATE command saves analytic workspace changes to disk.

For the complete syntax for the UPDATE command, see the OLAP DML Reference. The simplified syntax for the UPDATE command is show below.

```
UPDATE [dbname1 [dbname2 . . .]]
```

A *dbname* argument specifies the name of a read / write analytic workspace that is attached to your OLAP Services session. If you do not specify any analytic workspace names, then all the attached read / write analytic workspaces, including `express.db`, are updated.

For example, you can issue the following command to save all analytic workspace changes made so far in a session, including changes to `express.db` if it is attached as a read / write analytic workspace.

```
update
```

Updating shared analytic workspaces

If you have attached a shared analytic workspace and another user has read/write access, then that user's UPDATE 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.

Minimizing Analytic Workspace Growth

Ways to minimize analytic workspace growth

This guide presents some very basic and simple information about ways in which you can minimize analytic workspace growth.

You can minimize analytic workspace growth through the judicious use of NA stored pages and by frequently updating the analytic workspace when you are attached exclusively. You can completely reorganize the analytic workspace by exporting and importing all of the analytic workspace files.

Definition: NA pages

An NA page is an analytic workspace page that contains only NA values for a variable. Depending on the status of the variable at the time the NA values are assigned, NA pages are either unstored or stored:

- NA unstored pages — In most cases, if an analytic workspace page would contain only NA values for a variable, then the page is not actually stored. Instead a marker is stored. This marker indicates that the page would contain only NA values. These virtual analytic workspace pages are called NA unstored pages.
- NA stored pages — An NA stored page is an actual analytic workspace page that contains only NA values. Typically, you want NA stored pages to be created only if you want to reserve space for values of an in-place variable.

When are NA stored pages created?

NA stored pages are created only if all of the following conditions are true:

- The NA values are explicitly assigned.
- The variable is defined as an in-place variable when the NA values are assigned.
- The analytic workspace that contains the variable is attached in read/write, exclusive mode when the NA values are assigned.
- The variable contains, or has previously contained, at least one non-NA value.
- The NA values are distributed so that some analytic workspace pages contain only NA values.

Retrieving the number of NA pages

You can use the OBJ function with the NAPAGES keyword to retrieve the number of NA pages (either stored and unstored) in an analytic workspace.

You can also run DBREPORT to retrieve information about the number of NA pages (both stored and unstored) in an analytic workspace.

For more information on the OBJ function and the DBREPORT program, see the topic for the function or the program in the OLAP DML Reference.

Releasing NA stored pages

You can use the NAPAGEFREE command to release any NA stored pages that have been created for a variable. Once these pages are released, they can be used to store new data. NAPAGEFREE loops through all allocated pages for the variable and converts any NA stored pages into NA unstored pages. When it is finished, it reports the number of pages that have been freed.

For more information on the NAPAGEFREE command, see the topic for the function in the OLAP DML Reference.

Example: Releasing NA stored pages

The following example uses OBJ(DISKSIZE) to query the variable SALES before and after NAPAGEFREE is issued to show its reduction by the number of pages that are freed by NAPAGEFREE (three in this example).

The first OBJ function shows that 35 pages are being used to store the SALES variable.

```
show obj(disksize 'sales')
35
```

Now the NAPAGEFREE command frees three pages that contained only NA values.

```
napagefree sales
3 pages freed for SALES.
```

When the OBJ function is reissued, it shows that only 33 pages are now being used to store the SALES variable.

```
show obj(disksize 'sales')
32
```

When are unused pages released?

When many users are attached to an analytic workspace, unused pages are not actually released when the analytic workspace is updated. Instead, an erase list is created. The erase list identifies the pages that it can release later when only one user is attached to the analytic workspace.

Updating an analytic workspace when you are attached exclusively

When you update an analytic workspace and no other users are attached to the analytic workspace, the erase list is flushed and all unused pages are released. This creates more space in the analytic workspace files for new data. Consequently, to minimize analytic workspace growth, you want to update the analytic workspace frequently when you have exclusive use it.

Using EXPORT and IMPORT to minimize analytic workspace size

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. This procedure removes extra space. The new files may be substantially smaller.

To reorganize your analytic workspace by exporting and importing OLAP DML 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 command to update the new analytic workspace.
6. After checking that the objects were successfully moved into the new analytic workspace, delete the original analytic workspace.
7. Rename the new analytic workspace with the original name.

For more information on importing and exporting analytic workspace files, see the topics for the EXPORT and IMPORT commands in the OLAP DML Reference.

You cannot export and import SEGWIDTH specifications

If you use CHGDFN SEGWIDTH to specify the segment size of any variable, you should be aware that this information cannot be exported and imported. If you export any variable, when that variable is imported, it will use the default segment size.

Sharing Analytic Workspaces

Sharing analytic workspaces across sessions

An analytic workspace can be accessed simultaneously by several sessions. However, only one session can have the analytic workspace open with read/write access at any given time.

When you attach an analytic workspace, your default access to it is read-only. OLAP Services 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 updates 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 DATABASE command. If you request read/write access to an analytic workspace that is being used in read/write mode by another session, whether or not OLAP Services waits for the analytic workspace and the message OLAP Services returns to the

application depends on how you have coded the DATABASE command as described in "Waiting for an analytic workspace" on page 2-18.

Waiting for an analytic workspace

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 DATABASE 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 OLAP Services places you on the wait list for the analytic workspace. The number of seconds that OLAP Services waits for access depends on the value of the DefaultDBWaitTime setting. Use the OLAP Services Instance Manager to change the value of the DefaultDBWaitTime setting.

Strategies for attaching analytic workspaces in exclusive mode

If your analytic workspaces are in use almost all of the time, you need to develop a strategy for attaching the analytic workspace in read / write, exclusive mode. Some possible strategies are listed below:

- "Officially" schedule a time to attach the analytic workspace exclusively.
- Run a batch job every 30 minutes or so that attempts to attach the analytic workspace in read / write, exclusive mode and, once it is successful, updates the analytic workspace. You can use the Persistent Session and Command line utilities to set up batch jobs.

Command line utilities

You can submit batch jobs using the OLAP Service Manager (`xscosvc`). You can define scripts that invoke `xscosvc`. You can schedule the running of these scripts with the Unix `cron` facility, the Windows NT `at` command, or the job-scheduling facility within Oracle Enterprise Manager. For more information about input files, refer to the INFILE command in the OLAP DML Reference.

Working with AUTOGO Programs

What is an AUTOGO program?

You can create programs that are automatically executed when you attach an analytic workspace. When you attach an analytic workspace by using the DATABASE command without the ATTACH keyword, the workspace dictionary is searched for a program named AUTOGO. If it exists, then the program is executed before commands are accepted. You can use the NOAUTOGO keyword to specify that the AUTOGO program should not be executed.

You do not have to name a program AUTOGO to have it automatically execute when you attach an analytic workspace. You can use the AUTOGO keyword with the DATABASE command to specify that a program will be automatically executed with some name other than AUTOGO when you attach an analytic workspace. Even if a program named AUTOGO exists in the analytic workspace, the program you specify is still used after the AUTOGO keyword instead.

Analytic workspace permission programs are executed before any AUTOGO program that is associated with the analytic workspace is executed. For more information on permission programs, see "Adding Security to an Analytic Workspace" on page 2-20. For information on writing and debugging OLAP DML programs, see Chapter 8 and Chapter 9.

Example: AUTOGO program

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.

The `analysis` analytic workspace has the following AUTOGO program, which attaches the other two analytic workspaces.

```
database attach expense after analysis
database attach revenue after analysis
```

Running AUTOGO programs

Suppose that you write a program named ATTACH_DBS that attaches the analytic workspaces you want an application to use. You can run it when attaching the main analytic workspace, called `analysis`, with the following command.

```
database autogo attach_dbs analysis
```

If you named the program AUTOGO, you could run the program automatically with the following command.

```
database analysis
```

Adding Security to an Analytic Workspace

Types of security

You can protect analytic workspaces with a password and analytic workspace permission programs. When an analytic workspace is password-protected, users cannot attach it without specifying the password. When you provide an analytic workspace permission program for an analytic workspace, that program associates access rights with OLAP DML objects.

Assigning a password

At any time after you create an analytic workspace, you can assign a password to it by using the DATABASE command with the PASSWORD keyword. This command assigns a password to the current analytic workspace; if the current analytic workspace already has a password, then it replaces the old password with the new one.

Passwords can consist of up to 16 characters. They must begin with a letter or an underscore and can contain letters, numbers, periods (.), and underscores (_). Choose a password you can remember easily. Once you specify a password, you cannot access the analytic workspace without it.

A password does not become effective until you update the analytic workspace. Thereafter, you can attach that analytic workspace only if you supply this password in the DATABASE command.

Example: Assigning and using a password

The following command assigns the password `goldfinch` to the current analytic workspace (called `sales`).

```
database password goldfinch
```

To access the analytic workspace after this command is executed, you must, as shown below, use the password `goldfinch`.

```
database sales goldfinch
```

Removing a password

To remove a password from the current analytic workspace, use the DATABASE command with the PASSWORD keyword without specifying the *password* argument. Once you update the analytic workspace, the password is no longer required to attach the analytic workspace.

Using analytic workspace permission programs

When a user attaches an analytic workspace, the analytic workspace is checked to see if it contains a program called PERMIT_READ or PERMIT_WRITE. You do not have to create these programs; however, if they are present, then they are automatically executed when the user attaches an analytic workspace.

IF the user attaches an analytic workspace with . . .	THEN the following program is executed, it exists . . .
read-only access,	PERMIT_READ program.
read / write access,	PERMIT_WRITE program.

If you have analytic workspace 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 take special care when you edit them or use them in any other way, to ensure that you access the one in the appropriate analytic workspace.

Analytic workspace permission programs are executed before any AUTOGO program that is associated with the analytic workspace is executed. If a user specifies a password when attaching the analytic workspace, then the password is not immediately compared to the stored password that was specified with DATABASE PASSWORD. Instead, the password is passed as an argument to the analytic workspace permission program for processing.

Creating and designing analytic workspace 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 and the values of the permission conditions on which permission is

based. You write these programs as user-defined functions that return a Boolean value.

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

For information on writing and debugging OLAP DML programs, see Chapter 8 and Chapter 9.

Levels of access you can control using permission programs

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 or is not restricted to the access to specific objects or sets of object values. **Note:** All of the objects referred to in a given permission program must exist in the same analytic workspace.

Using PERMIT commands to restrict access

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.

You can specify permission to access OLAP DML objects with PERMIT commands. The PERMIT command can use permission conditions based on values that are returned by the SYSINFO function. In this manner, you can specify permission based on the user ID under which the session is running or the groups to which the user ID belongs.

Making an analytic workspace read-only

To protect an analytic workspace from inadvertent changes, you should ensure that users attach the analytic workspace in read-only (RO) mode unless you know that

users need to make permanent changes in the analytic workspace. By default, an analytic workspace is read-only when it is attached. You can also explicitly make the analytic workspace read-only at the system level.

Users can use a read-only analytic workspace in the same way as an ordinary analytic workspace; users can even make changes to it during a session. However, users cannot make the changes permanent on disk by updating. The UPDATE command has no effect on an analytic workspace with read-only access. This protects data you do not want users to change.

Obtaining Analytic Workspace Information

Viewing a complete analytic workspace description

The DBDESCRIBE 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 OLAP DML object.
- A list of OLAP DML objects that are sorted alphabetically.
- Detailed descriptions of all OLAP DML 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 DBDESCRIBE is frequently very long, you can direct it to a file with the OUTFILE command.

```
outfile 'filename'
dbdescribe
outfile eof
```

Obtaining general analytic workspace information

The DATABASE function returns various kinds of information about attached analytic workspaces. For example, you can use the DATABASE function to learn your read or write access rights to an analytic workspace or to determine if an attached analytic workspace has been changed by a user with read/write access.

For the complete syntax for the DATABASE function, see the OLAP DML Reference. The simplified syntax of the DATABASE function is shown below.

```
DATABASE(choice [database-name])
```

The keyword you specify for *choice* determines the type of information that is returned by the DATABASE function. Examples of keywords are: ATTACHED, CHANGED, FILESIZE, 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 DATABASE(NAME) eq 'MYSALES'
     then report sales.m
     else report gensales
```

Listing the names of 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).

Example: Listing the names of all of the objects in an analytic workspace

If the demo analytic workspace is attached, issuing the command `listnames` produces the following output.

```
11 DIMENSIONs 19 VARIABLEs   4 RELATIONs      2 VALUESETs
------------- -------------- -----------      ----------
CHOICE        ACTUAL         DIVISION.PRODUCT PRODUCTSET
DISTRICT      ADVERTISING    MARKET.MARKET    QUARTERSET
DIVSION       BUDGET
LINE          CHOICEDESC
MARKET        DEMOVER
MARKETLEVEL   EXPENSE
MONTH         FAST
MONTH         INDUSTRY.SALES
PRODUCT       NAME.LINE
QUARTER       NAME.PRODUCT
REGION        NATIONAL.SALES
YEAR          PRICE
              PRODUCT.MEMO
              SALES
              SALES.FORECAST
              SALES.PLAN
              SHARE
              UNITS
              UNITS.M
```

Viewing the definitions of OLAP DML objects

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

```
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.

Listing objects that are dimensioned by a specific dimension

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

Example: Listing objects that are dimensioned by a specific dimension

For example, to find out which objects in the demo analytic workspace 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 database DEMO
-------------------------------------------------------------
ACTUAL          ADVERTISING        BUDGET
EXPENSE         FCST               NATIONAL.SALES
PRICE           PRODUCT.MEMO       SALES
SALES.FORECAST  SALES.PLAN         SHARE
UNITS           UNITS.M
```

Obtaining information about OLAP DML objects

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

Example: Obtaining the number of dimensions for the variable

The following command obtains the number of dimensions for the variable UNITS in the demo analytic workspace. The output is shown below the command.

```
show obj(numdims 'units')
3
```

Example: Obtaining the data type of a variable

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

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

Obtaining information about groups of objects

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. The NAME dimension contains the names of all the objects that are defined in the analytic workspace.

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.

Example: Identifying objects by their dimensions

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
```

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
the status of a dimension,	Chapter 6 the entry for the STATUS command in the OLAP DML Reference
limiting dimensions,	Chapter 6 the entry for the LIMIT command in the OLAP DML Reference
the DBDESCRIBE program, LISTBY command, LISTNAMES program, NAME dimension, OBJ function	the entry for the program, command, or function in the OLAP DML Reference

3

Defining Data Objects

Chapter summary

This chapter introduces the OLAP DML data structures. It explains how to define OLAP DML objects and change the definition of those objects.

List of topics

This chapter includes the following topics:

- Overview: Defining OLAP DML objects
- Defining Dimensions
- Defining Relations
- Defining Variables
- Defining Variables That Handle Sparse Data Efficiently
- Defining Hierarchical Dimensions and Variables That Use Them
- Defining Metadata
- Changing the Definition of an Object

Overview: Defining OLAP DML objects

What are object definitions?

It is important to understand the distinction between an object's definition and its data. An object's definition is its description in the workspace dictionary of the analytic workspace. An object's data 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 (stored procedures), do not have data.

Defining OLAP DML objects using OLAP DML commands

Once you have created an analytic workspace, you can begin defining OLAP DML 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 [DATABASE dbname]
```

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

Important: Because each analytic workspace has its own dictionary of OLAP DML objects, you can define objects with the same name in more than one analytic workspace. However, to prevent unexpected results, you should take care to provide unique names for objects in separate analytic workspaces that will be active at the same time.

The *object-type* argument specifies the type of OLAP DML object that is being defined. The default is VARIABLE. You can specify any of the valid object types as outlined in "OLAP DML objects you define using the DEFINE command" 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.

The DATABASE *dbname* phrase specifies the name of an attached analytic workspace in which you want to define the object. If you do not specify an analytic workspace name, then the current analytic workspace is used.

For the complete syntax for the DEFINE command, see the entry for the command in THE OLAP DML Reference.

OLAP DML objects you define using the DEFINE command

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

Object Type	Description
DIMENSION	An object that 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 analytic workspace.
RELATION	An object that establishes a correspondence between the values of a given dimension and the values of that dimension or other dimensions in the analytic workspace.
VARIABLE	An object that stores data. The data type of a variable indicates the kind of data that it contains.
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.
FORMULA	An object that represents a stored calculation, expression, or procedure that produces a value.
MODEL	An object that 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	An object that contains a series of OLAP DML commands. A program is a stored procedure that executes a set of related commands.
VALUESET	An object that contains a list of dimension values for a particular dimension.
AGGMAP (AGGREGATION MAP)	An object that contains a set of interrelated commands that are used to specify which data in a variable should be aggregated (with the AGGREGATE command) and which data should be calculated on the fly (with the AGGREGATE function).

Defining Dimensions

Definition: Dimension

A dimension is an OLAP DML object that holds a list of values that provide the organization for one or more OLAP DML variables. A dimension value is similar to a key in a relational analytic workspace; it acts as an index to data. 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 Feb98, Mar98, and Apr98.

Dimension values let you identify your data and provide an easy way to target the data you need for a particular purpose. When your users display or analyze your data, they select the values to work with.

Types of dimensions

OLAP DML supports both flat and hierarchical 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 that you can then use to dimension a single variable that contains data for all the levels. Some dimensions have multiple hierarchies based on them. For more information on hierarchical dimensions, see "Defining Hierarchical Dimensions and Variables That Use Them" on page 3-20.

Note: When you define a variable using the SPARSE keyword, an internal object called a *composite* is automatically created. A composite shares some of the characteristics of a dimension and contains values that are combinations of values in other dimensions. For more information on composites, see "Defining Variables That Handle Sparse Data Efficiently" on page 3-15.

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 need to define a hierarchical dimension.

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.

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, it stores each new value at the end of the array.

Example: How data for simple flat dimensions is stored

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 values are actually stored as shown below.

PRODUCT Dimension			
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 dimension array looks like the following figure.

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

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
adding values to dimensions,	Chapter 5.
using dimensions in expressions,	Chapter 4.
reports,	Chapter 12.
defining dimensions,	the entry for the DEFINE DIMENSION command in THE OLAP DML Reference.

Defining Relations

Definition: Relation

A relation is an OLAP DML 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 two relations, or a self-relation. Additionally, relations between dimensions in your analytic workspace that have time data types (DAY, WEEK, MONTH, QUARTER, or YEAR) are automatically defined.

How relations are dimensioned

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

Dimensioning with the larger 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.

Dimensioning with the fewer number of values

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. Dimension values vary in the order you list them in the definition, with the first value varying fastest and the last value varying slowest.

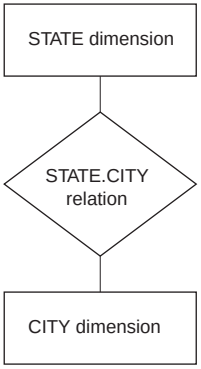
The data values that are stored for a relation are the indexes of the related dimension.

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.

STATE.CITY Relation			
CITYPosition (Value)	C1 (Atlanta)	C2 (Chicago)	C3 (Springfield)
STATEPosition (Value)	S1 (Georgia)	S2 (Illinois)	S2 (Illinois)

Example: Relation between two dimensions

Most relations are a single-dimensional array that relates the values of one dimension with another. For example, as the figure below illustrates, 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.

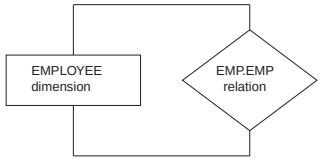
```
STATE
--------------
GEORGIA
ILLINOIS

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's 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.

For information about using self-relations with hierarchical dimensions, see "Defining Hierarchical Dimensions and Variables That Use Them" on page 3-20.

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
using self-relations with hierarchical dimensions,	"Defining Hierarchical Dimensions and Variables That Use Them" on page 3-20.
adding values to relations,	"Assigning Values to Data Objects" on page 5-13.
using relations in expressions,	"Using OLAP DML Objects in Expressions" on page 4-6.

Defining Variables

Definition: Variable

A variable is an OLAP DML 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, inplace, or temporary. You can also define variables in programs, as described in "Defining local variables" on page 8-9.

Permanent variables

A permanent variable is a variable for which both the variable's values and definitions are stored on disk. The values of permanent variables are written to new pages in the analytic workspace as you make changes to the values of the variable. However, the stored values of the permanent variable are not actually changed when an UPDATE command is processed for the analytic workspace that contains the variable. Consequently, if an update of an analytic workspace is unsuccessful, then the original values of the permanent variable can be retrieved.

Inplace variables

Like permanent variables, both the values and definitions of inplace variables are stored on disk. The way that inplace variables are updated depends on how the analytic workspace is attached:

- When the analytic workspace is attached in read/write, non-exclusive mode, inplace variables are updated in the same way that permanent variables are updated.
- When the analytic workspace is attached in read/write, exclusive mode, the values of an inplace variable are stored whenever it writes pages to the disk. Additionally, when you change the values of inplace variable, the new changed values are stored over the inplace variable's old values. Consequently, if an update of an analytic workspace is unsuccessful, then the original values of an inplace variable might not still be stored in the analytic workspace, and it might not be possible to restore the variable's original values.

For more information on attaching analytic workspaces, see "Gaining Access to a Workspace from OLAP Worksheet" on page 2-4.

Temporary variables

For more efficient use of disk space, the OLAP DML also lets you define temporary variables that have values only during the current OLAP Services session. When you update the analytic workspace, only the definitions of temporary variables are saved. When you exit from the analytic workspace, the data values are discarded.

When to use inplace variables

There are both advantages and disadvantages to using inplace variables:

- Advantages of inplace variables — Once the values for an inplace variable have been stored, new analytic workspace pages are no longer created to store new, changed values. Consequently, the major advantage of using an inplace variable is that the analytic workspace that contains the variable grows at a slower rate and has fewer free pages then the analytic workspace would have if you defined the variable as a permanent variable.
- Disadvantages of inplace variables — The major disadvantages of using inplace variables rather than a permanent variables are:
  - An inplace variable is always in an indeterminate state while it is being updated. Because the old values are overwritten when the new ones are stored, there is no way to know exactly what data is stored in the variable at

any given moment. Depending on the other processing that is being performed, an inplace variable that is being updated can be computationally inconsistent. The values of an inplace variable are in a determinate state only after an UPDATE command is processed for the analytic workspace that contains the variable.

- If an UPDATE command for an analytic workspace that contains an inplace variable fails, it might not be possible to restore the original values of the inplace variable if its stored values have been overwritten. Instead, you must explicitly restore the inplace variable's original values.

Recommendations for using inplace variables

Oracle Corporation recommends that you only use inplace variables if you can guarantee that the following conditions are met:

- You can reconstruct the data in the inplace variable if the update of the analytic workspace that contains the variable fails.
- You are the only user accessing the analytic workspace when you update the inplace variable, for example, when you explicitly attach an analytic workspace in read / write, exclusive mode as described in "Specifying the analytic workspace attachment mode" on page 2-11.

How variable data is stored

The order in which you list the dimensions in a variable's definition determines how that variable's data will be stored and accessed. The first dimension that you list in the variable definition is referred to as the *fastest-varying dimension*, and the last dimension that you list is referred to as the *slowest-varying dimension*.

Example: How variable data is stored

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>
```

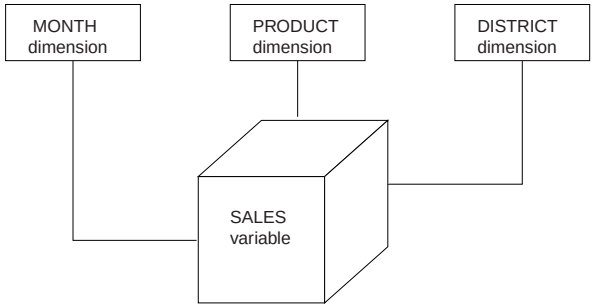
The data for a multidimensional variable is stored as a linear stream of values, in which the values of the fastest-varying dimension are clustered together. For example, for the OPCOSTS variable, the values for Boston for all the months are

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, as shown below.

OPCOSTS variable						
Dimension Values	JAN97 Boston	FEB97 Boston	. . . . .	JAN97 Chicago	FEB97 Chicago	. . . . .
Variable Values	16000.77	16000.28	. . . . .	19000.21	19000.24	. . . . .

Example: Three-dimensional variable

The demo analytic workspace contains the SALES variable, which is a three-dimensional array dimensioned by MONTH, PRODUCT, and DISTRICT.



Assume that the MONTH, PRODUCT, and DISTRICT dimensions have 36, 5, and 6 values, respectively, and that the SALES variable has the following definition.

```
define sales variable decimal <month product district>
```

The SALES variable contains 1,080 cells, which is the total number of cells in each dimension multiplied together or, in this case, 36 times 5 times 6.

```
DISTRICT: BOSTON
                        -------------------SALES-------------------
                        -------------------MONTH-------------------
PRODUCT         JAN96       FEB96       MAR96
--------------- ----------- ----------- ----------- -----------
TENTS                50,808.96      34,641.59       45,742.21       . . .
CANOES               70,489.44      82,237.68       97,622.28       . . .
RACQUETS             56,337.84      60,421.50       62,921.70       . . .
SPORTSWEAR           57,079.10      63,121.50       67,005.90       . . .
FOOTWEAR             95,986.32      101,115.36     103,679.88       . . .
```

```
DISTRICT: ATLANTA
                    -------------------SALES-------------------
                    -------------------MONTH-------------------
PRODUCT       JAN96      FEB96      MAR96
------------- ---------- ---------- ---------- ----------
TENTS               46,174.92       50,553.52      58,787.82       . . .
CANOES              56,271.40       61,828.33      77,217.62       . . .
  .
  .
  .
```

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
populating variables,	“Assigning Values to Data Objects” on page 5-13
using variables in expressions,	“Using OLAP DML Objects in Expressions” on page 4-6
defining variables,	the entry for the DEFINE VARIABLE command in THE OLAP DML Reference

Defining Variables That Handle Sparse Data Efficiently

Definition: Sparse data

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

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 past months in the MONTH dimension, but the data is NA.
- *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 some of those products and other ones, too.

Definition: Composite

A composite is an internal object that is used to compactly store a variable with sparse data. 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 an OLAP DML object; it is merely an internal structure. If you define a variable you use the SPARSE keyword to request that an unnamed composite is automatically created.
- A named composite is an OLAP DML object that is you define using the DEFINE COMPOSITE command. Later, when you are defining or accessing a variable you use this name along with the dimension names from which it is built.

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

In general, why you should use composites

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.

Specifically, why you should use named composites

Using a named composite in the variable's dimension list tells OLAP Services that those dimensions in the named composite are sparse dimensions on this variable, and that this composite is shared only with other variables that use the same named composite.

In general, using named composites is a good practice. This is because any variables that are defined with an unnamed composite and 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. Using named composites makes it easier to track which variables share the same composite.

Note: 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

To ensure that a variable uses a minimum of disk storage space, when you define a multidimensional variable, you can specify that a composite is used to store the data for one or more of the variable's dimensions.

First, define a named composite as an OLAP DML object by using the DEFINE COMPOSITE command. Then, define the variables by using the following syntax to include a named composite in each variable's dimension list.

```
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.

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.

What happens when you add data to a variable that uses a composite

When you define a multidimensional variable, you can specify that a composite is used to store the data for one or more of the variable's dimensions. Later, as you

add data to the variable's dimensions for which you defined a composite, the following actions are taken:

1. The composite is filled with dimension-value combinations.
2. The data for the variable is stored using the composite's 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's 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: Defining a variable that uses a named composite

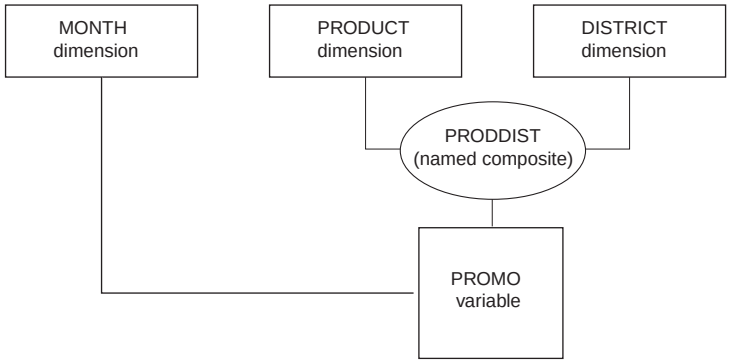
If your company does promotional marketing for certain products in some but not all districts, then your variable data will be sparse along the PRODUCT and DISTRICT dimensions. Therefore, suppose you define a composite named PRODDIST, whose base dimensions are PRODUCT and DISTRICT. There are dimension-value combinations in the composite only for those values that have data. For example, if you ran a promotion for tents but not canoes, then the composite will include 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 conceptual figure below 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 is an example of the sequence in which the data for the PROMO variable might be stored.

Sequence in which 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 that dimension's values.

Example: Defining a variable with a single-dimension composite

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 ORG, but 90 percent of the LINE dimension cells will have NA values.

To handle sparse data in this case, when you define BUDGET, specify a composite for only the LINE dimension as shown below.

```
define budget decimal <sparse <line> division month>
```

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
working with sparse data,	"Working with NA Values" on page 4-40
using composites in expressions,	"Using composites in expressions" on page 4-7
defining composites,	the DEFINE COMPOSITE command in the OLAP DML Reference
defining conjoint dimensions,	the DEFINE DIMENSION command in the OLAP DML Reference
defining variables that use composites,	the DEFINE VARIABLE command in the OLAP DML Reference

Defining Hierarchical Dimensions and Variables That Use Them

Definition: Hierarchical dimension

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 based on them.

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.

Example: Hierarchical dimension values

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 that uses 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.

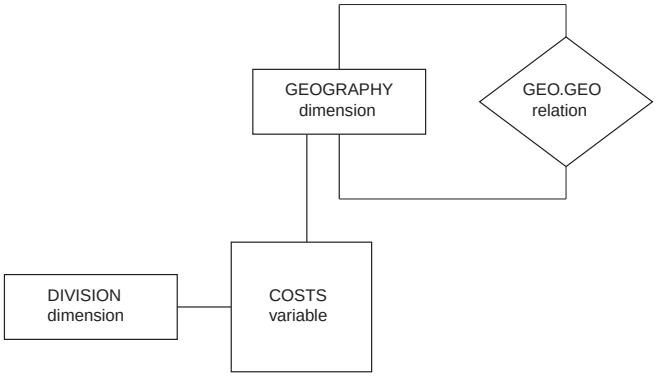
Frequently, the cells in the variable that correspond to upper level values in the hierarchical dimension contain the sum or total of the values in the variable's cells that correspond to the lower level dimension values. For example, in a SALES variable that is defined with a TIME dimension, the variable's cells that correspond to each quarter 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. For more information on aggregating data, see "Aggregating Data" on page 5-19.

Example: Hierarchical dimension and variable that uses it

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>
```

The figure below illustrates the values of a self-relation called GEO.GEO that is defined to assign cities to regions.

```
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
```

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
Defining dimensions,	the entry for the DEFINE DIMENSION command in the OLAP DML Reference
adding values to dimensions,	“Maintaining Dimensions and Composites” on page 5-3
using dimensions in expressions,	“Using dimensions in expressions” on page 4-6
aggregating data,	“Aggregating Data” on page 5-19

Defining Metadata

Definition: Metadata for OLAP DML objects

Metadata is a group of data objects that describes the OLAP DML objects that you define.

Why and how to use metadata

The purpose of metadata is to provide you with the flexibility to specify how the OLAP DML objects will be displayed in applications. Therefore, you should use metadata when the way in which OLAP DML objects are displayed matters.

You must define metadata and set its property to YES in order to use that metadata.

Using the metadata appendix

This guide provides an appendix that describes all of the metadata that you can define and use. Refer to Appendix A to learn the following:

- What metadata is available — You may or may not need to define all possible types of metadata, depending on your needs. Read the appendix thoroughly in order to understand what kind of metadata is possible to define, then decide which metadata you need.
- The purpose of each metadata object — Each metadata object serves a specific purpose. The description of each metadata object explains its purpose and use.
- How to define metadata — The description of each metadata object includes the syntax for defining that metadata object, as well as an example.
- How to set the metadata's property to YES — The description of each metadata object includes an example of how you set its property to YES, which is what makes that object functional.

Changing the Definition of an Object

When can you change 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.

Commands that you can use to make changes to an object definition

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

Command	Description
EQ	Allows you to specify the expression to be calculated for a formula that has already been defined
EXTARGS	Assigns arguments to the definition of an EXTCALL object
LD	Assigns a long description to an object definition
MODEL	Allows you to enter completely new contents into a new or existing model
PERMIT	Assigns access permission to an object definition
PROGRAM	Allows you to enter completely new contents into a new or existing program
PROPERTY	Assigns a property to an object definition
VNF	Assigns a value name format to the definition of a time dimension

Example: Changing the definition of a variable

Suppose that you have defined a Boolean variable named ONPLAN. Later, you want to add a description to the variable's definition.

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 being tracked on a special plan?
```

Example: Changing the storage type of a variable

You can redefine the access mode of a variable by using the CHGDFN command, which is shown below.

```
CHGDFN varname INPLACE
               PERMANENT
```

For more information on the DEFINE and CHGDFN commands, see the topic for the command in the OLAP DML Reference.

4

Working with Expressions

Chapter summary

Expressions represent data values in the grammar of the OLAP DML. This chapter explains how to create and use expressions.

List of topics

This chapter includes the following topics:

- OLAP DML Data Types
- Using OLAP DML Objects in Expressions
- OLAP DML Operators
- Introducing Expressions
- Expressions and Dimensionality
- Specifying a Single Value for the Dimension of an Expression
- Using Functions in Expressions
- Numeric Expressions
- Text Expressions
- Boolean Expressions
- Conditional Expressions
- Substitution Expressions
- Working with NA Values

OLAP DML Data Types

Basic data types

OLAP DML data types fall into five categories which are referred to as basic data types and are described in the following table.

Basic Type	Specific Type
Numeric	INTEGER, SHORTINTEGER, DECIMAL, SHORTDECIMAL
Text	TEXT, ID
Boolean	BOOLEAN
Date	DATE
Time	DAY, WEEK, MONTH, QUARTER, YEAR

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, TEXT, ID, BOOLEAN, and DATE data types are supported.
- For dimension values, the INTEGER, TEXT, ID, DAY, WEEK, MONTH, QUARTER, and YEAR 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 ±2,147,483,647).
SHORTINTEGER	A whole number (in the range of ±32,767.
DECIMAL	A decimal number (with up to 15 significant digits).
SHORTDECIMAL	A decimal number (with up to 7 significant digits).

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.

Examples of literal numeric values

Examples of literal numeric values are:

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

Text data types

The following types of text data types are supported.

Data Type	Data Value
TEXT	Any number of alphanumeric characters enclosed in single quotes (’).
ID	Up to 8 alphanumeric characters enclosed in single quotes (’).

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	Linefeed
\r	Carriage return
\t	Horizontal tab
\"	Double quote
\’	Single quote
\\	Backslash
\dnnn	Character with ASCII code nnn decimal, where \d indicates a decimal escape and nnn is the decimal value for the character
\xnn	Character with ASCII code nn hexadecimal, where \x indicates a hexadecimal escape and nn is the hexadecimal value for the character

Examples of literal text values

Examples of literal text values are:

```
'First Quarter\'s Earnings'
'sales data eif'
'NONE'
'\n'
'JAN96'
'c:\\plan97\\budget.db'
```

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

By default, Boolean values as YES and NO are displayed. However, you can use the NOSPELL and YESSPELL functions to specify other values for display.

Working with Boolean expressions is discussed in "Boolean Expressions" on page 4-28.

DATE data type

A DATE data type is provided that you can use to represent date values. Dates range from January 1, 1000, to December 31, 9999.

To control how values are formatted with the DATE data type in output, use the DATEFORMAT option. For more information on using the DATEFORMAT option, see the entry for the option in the OLAP DML Reference.

To represent DATE values, specify them in either quoted text or integer format.

Representing DATE values as quoted text

For quoted text, specify a group of characters enclosed in single quotes (’). Format the text in one of the styles specified by the DATEORDER option and described in the following table.

IF the style is...	THEN specify...	Examples
numeric,	the day, month, and year as three integer numbers with one or more separators between them.	’24/4/97’ ’24-04-1997’
packed numeric,	the day, month, and year as three integer numbers with no separators between them.	’240497’ ’04241997’
month name,	the day and year as integer numbers and the month as characters.	’24APR97’ ’24 ap 97’ ’April 24, 1997’

See the entry for DATEORDER in the OLAP DML Reference for detailed rules for these styles.

Representing DATE values as integers

For an integer, specify a number between -328,717 and 2,958,464 (with the integer 1 corresponding to January 1, 1900).

Time data types

For dimensions, five time data types (DAY, WEEK, MONTH, QUARTER, and YEAR) are supported. You can specify a dimension value for a time dimension in either date or value name (VNF) format:

- To specify a value in date format, use any of the input styles listed for the DATEORDER option. You need to specify only the date components that are relevant for the data type that is defined for the dimension. If you specify a full date, the current value of the DATEORDER option is used to resolve any ambiguities.
- To specify a value in VNF format, use the formats outlined in the VNF command. A VNF is a template that controls the input and display format for values of a given time dimension. The template can include format specifications for any of the components that identify a time period (day, month, calendar year, fiscal year, and period within a fiscal year).

Note: You can use the MAKEDATE function to create a full date value from a day, month, and year. For more information on the MAKEDATE functions, see the entry for the function in the OLAP DML Reference.

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
functions that you can use to manipulate text, numbers, dates and time periods,	the OLAP DML Reference
specifying file names,	"Specifying File Names in the OLAP DML" on page 11-5

Using OLAP DML Objects in Expressions

Overview: Using OLAP DML objects in expressions

You can use OLAP DML objects and functions in expressions as described below:

- You can use a dimension, 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 subexpression 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 as described in "Assigning Values to Data Objects" on page 5-13.

Using dimensions in expressions

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

If the dimension has a data type of TEXT, then, in most cases, the dimension values are referenced as text values.

However, dimension values are referenced by their positions (integers) in the dimension array and uses the values numerically 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 position number is based on the default status list, not on the current status.

Note: When you use a dimension with a data type of DATE, if you want the dimension value to be treated as an integer position, then you must use the CONVERT function.

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 you specify its name.
- If the composite is unnamed, then you 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 variable's fastest-varying dimension 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 will 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 QDR to specify one value from each of the variable's dimensions.

For example, if the OPCOSTS variable is dimensioned by MONTH and CITY, specifying Jan95 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 OLAP DML 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 a variable

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 composite's base dimensions 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 1995. Since no coupons were issued in March 1995, the report displays NA in that column.

```
limit month to 'JAN95' 'FEB95' 'MAR95'
limit market to 'EAST'
limit product to 'SPORTSWEAR'
report coupons

MARKET: EAST
                ------------COUPONS-------------
                ------------MONTH--------------
PRODUCT           JAN95      FEB95      MAR95
--------------- ---------- ---------- ----------
SPORTSWEAR          1,000      1,000         NA
```

Changing the default behavior of looping commands

However, for performance reasons, you can change the default looping behavior for the REPORT, ROW, RETRIEVE, FETCH, and = commands so that those commands loop over the values in the composite rather than all of the base dimension values. For more information on these commands, see the entry for each command in the OLAP DML Reference.

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.

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
OLAP DML data objects,	Chapter 3
using OLAP DML data objects in assignment statements,	“Assigning Values to Data Objects” on page 5-13
user-defined functions,	“Writing User-Defined Functions” on page 8-16

OLAP DML Operators

Definition: Operator

An operator is a symbol that transforms a value or combines it in some way with another value.

Categories of operators

The OLAP DML operators fall into the categories described in the following table.

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 "OLAP DML arithmetic operators" on page 4-23.
Assignment	An operator that you use to create an assignment statement that stores the results of an expression into an OLAP DML object. For more information on using assignment statements, see "Assigning Values to Data Objects" on page 5-13.
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 "Boolean operators" 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 "Boolean operators" on page 4-29.
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 4-39.
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 4-37.

Introducing Expressions

Definition: Expression

Expressions represent data values in the grammar of the OLAP DML language. You can use expressions as arguments in commands or functions and as values for OLAP DML options. An expression often performs a mathematical or logical operation. It always evaluates to a result in one of the OLAP DML 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 "OLAP DML Data Types" on page 4-2.

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 OLAP DML 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 an expression's data type. 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 or ID data anywhere text is called for, and you can use integers and decimal numbers interchangeably.

OLAP DML data types are discussed in "OLAP DML Data Types" on page 4-2.

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 a OLAP DML 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.

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
functions,	the OLAP DML Reference
formulas,	Chapter 7 the entry for the DEFINE FORMULA command in the OLAP DML Reference
dimension status and limiting dimensions,	Chapter 6 the entry for the LIMIT command in the OLAP DML Reference
assigning permissions to dimension values,	"Adding Security to an Analytic Workspace" on page 2-20 the entry for the PERMIT command in the OLAP DML Reference
user-defined functions,	Chapter 8
qualified data references,	"Specifying a Single Value for the Dimension of an Expression" on page 4-16

Expressions and Dimensionality

How an expression is dimensioned

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	More Information
Variable Relation Formula	The dimensions listed in the object's definition	**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 4-16
Function	In most cases, the union of the dimensions of its input arguments	**Note 1:** 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. **Note 2:** The dimensions of a user-defined function depend on how it has been coded.

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.

For more information on the PARSE command and the INFO function, see the entry for the command or function in the OLAP DML Reference.

Example: Determining the dimensions of an expression

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.

Example 1: How dimension status affects the results of an expression

You can see the changes in the results reported by the TOTAL function as you change the status of PRODUCT and MONTH, both of which are dimensions of the SALES variable.

```
limit month to all
limit district to all
limit product to all
report width 22 total(sales product)
```

The output of this report command is shown below.

```
PRODUCT         TOTAL(SALES PRODUCT)
--------------- ----------------------
TENTS                   10,430,420.75
CANOES                  11,699,953.48
RACQUETS                13,550,445.01
SPORTSWEAR              14,910,328.52
FOOTWEAR                12,590,595.74
```

Example 2: How dimension status affects the results of an expression

The following commands produce the output shown below them.

```
limit product to 'RACQUETS' 'SPORTSWEAR'
report width 22 total(sales product)
PRODUCT         TOTAL(SALES PRODUCT)
--------------- --------------------
RACQUETS                13,550,445.01
SPORTSWEAR              14,910,328.52
```

Example 3: How dimension status affects the results of an expression

The following commands produce the output shown below them.

```
limit month to year 'YR96'
report width 22 total(sales product)
PRODUCT         TOTAL(SALES PRODUCT)
--------------- --------------------
RACQUETS                 6,957,866.18
SPORTSWEAR               7,703,196.64
```

Specifying a Single Value for the Dimension of an Expression

What is a QDR?

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 temporarily reference a value that is not in 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 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*.

Qualifying a complex expression

To qualify a complex expression, you should use the QUAL function. For more information, see the entry for QUAL.

Qualifying a variable

You can qualify any or all of a variable's dimensions 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 may 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.

Example: Temporarily limiting the dimension of a variable

In the demonstration analytic workspace, demo, 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 FETCH 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 fetch this data by using a QDR.

```
fetch sales(district 'BOSTON')/total(sales)
```

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(s) of the relation. The relation must be related to the variable's dimension you are qualifying, and it must be dimensioned by the replacement dimension.

Example: 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  JAN95    FEB95    MAR95    APR95    MAY95    JUN95
--------  -------- -------- -------- -------- -------- --------
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
------  ----- ------- ----- ------- ----- -------  ----- --------  ----- --------
JAN95   1.00  Hawley  0.82  Hawley  1.02  Carey    0.91  Musgrave  0.92  Musgrave
FEB95   0.84  Hawley  0.96  Hawley  1.00  Carey    0.80  Musgrave  1.07  Musgrave
MAR95   0.87  Jones   0.95  Jones   0.87  Carey    0.88  Musgrave  0.91  Musgrave
APR95   0.91  Jones   0.93  Jones   0.99  Carey    0.94  Musgrave  0.95  Musgrave
.
.
.
```

Example: Qualifying more than one dimension of a variable

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 JUN95, TENTS, and SEATTLE, use the following QDR.

```
fetch sales(month 'JUN95', product 'TENTS', district 'SEATTLE')
```

This command fetches the single value: 113,806.48.

Qualifying a relation

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

Example: Qualifying a relation

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 fetches the value: WEST.

```
fetch 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.

Example: Qualifying a dimension

The following expression specifies one value of DISTRICT, the one contained in the single-cell variable MYDISTRICT.

```
district(district mydistrict)
```

Using QDRs to assign a value to a specific cell of a data object

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.

Example: Using QDRs to assign a value to a specific cell of 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
```

Using ampersand substitution with QDRs

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.

Example: Using ampersand substitution with QDRs

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 explicitly 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: 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.
```

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
limiting dimensions,	Chapter 6
qualified data references,	"Specifying a Single Value for the Dimension of an Expression" on page 4-16
ampersand substitution,	"Substitution Expressions" on page 4-39
the QUAL function,	the entry for the function in the OLAP DML Reference

Using Functions in Expressions

Definition: Function

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.

Using functions in the OLAP DML

You can use a function wherever you need to use an expression in a command or function, or even within another expression. To include a function in an expression, specify the name of the function followed by its arguments enclosed in parentheses.

In addition to using the predefined OLAP DML functions, you can define a program that behaves like a function by returning a value.

Numeric Expressions

What is a numeric expression?

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

- Numeric literals
- Numeric variables or formulas
- Dimensions
- OLAP DML 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.

OLAP DML 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, which are summarized below.

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

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 INTEGER, SHORTINTEGER, DECIMAL, and SHORTDECIMAL 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 DECIMAL or SHORTDECIMAL,	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 4-28 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 Express stores NA.
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 Express stores NA.
assign the value of a decimal expression to a variable with the SHORTDECIMAL data type,	only the first 7 significant digits are stored.

If these conversions are not what you want, then you can use OLAP DML 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. When you use a dimension with a data type of DATE, you must use the CONVERT function when you want the dimension value to be treated as an integer position.

For example, the MONTH dimension in the `demo` analytic workspace has JAN95 in position 1, FEB95 in position 2, and so on. Even when the list is sorted alphabetically so that APR95 is first, the value APR95 remains in position 4.

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 or subtract a number from a time period,	the time period at the appropriate interval in the future or the past, similar to 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 dimension's default status list.

Limitations of dates in expressions

The following list outlines the ways in which you *cannot* use dates in expressions.

- You cannot add two dates together.
- You cannot add or subtract a literal value from a dimension value with a date data type. Both operands must actually be dimension values.

  For example, suppose that M1 is a dimension with a data type of MONTH. An error message is returned when you attempt to subtract the literal value AUG97

from the first value in status for the M1 dimension by issuing the first command shown below. However, the number of months between the two values is displayed when, as shown in the second command, you use a QDR to identify AUG97 as a value of M1 and then subtract this dimension value from the first value in status for the M1 dimension.

```
Incorrect: show m1 - 'AUG97'

Correct: show m1 - m1(m1 'AUG97')
```

- You cannot specify time periods that have different phases or lengths in the same calculation.

  For example, if you tried to subtract a week from a month, then the result would not have any meaning. If you need to compare time periods of different types, then use the IN operator.

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 may want to use the DECIMAL data type for variables that have a fractional decimal component, such as sales, costs, and other 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 4-28 for information on how to handle such comparisons.

Controlling errors during calculations

You can control the following types of errors:

- Division by zero — 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 divide an NA value by zero, then the result is NA; no error occurs. 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

What is a text expression?

A text expression evaluates to data with either the TEXT 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)`

Example: Text expression

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

The following commands produce the output shown below them.

```
show obj(type 'textvar')
VARIABLE
```

The following commands produce the output shown below them.

```
show obj(type textvar)
DIMENSION
```

Working with dates in text expressions

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

The current template in the DATEFORMAT option is used to format the text. If you want to override the current DATEFORMAT template, then you can convert the DATE value to text by using the CONVERT function with a date-format argument. See the entry for the CONVERT function in the OLAP DML Reference for an example.

Once a DATE value is stored in a text variable, the DATEFORMAT template is no longer used to format the display of the value, and subsequent changes to DATEFORMAT have no impact.

Boolean Expressions

What is a Boolean expression?

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, but you can change the output by setting the YESSPELL and NOSPELL options.

Boolean operators

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.

Operator	Operation	Example	Priority
NOT	Returns opposite of Boolean expression	`NOT(YES) = NO`	1st
EQ	Equal to	`4 EQ 4 = YES`	2nd
NE	Not equal to	`4 NE 4 = NO`	2nd
GT	Greater than	`5 GT 7 = NO`	2nd
LT	Less than	`5 LT 7 = YES`	2nd
GE	Greater than or equal to	`8 GE 8 = YES`	2nd
LE	Less than or equal to	`8 LE 9 = YES`	2nd
IN	Is a date in a time period?	`'1JAN97' IN WI.97 = YES`	2nd
LIKE	Does a text value match a specified text pattern?	`'EXPRESS' LIKE '%PRE%' = YES`	2nd
AND	Both expressions are true	`8 GE 8 AND 5 LT 7 = YES`	3rd
OR	Either expression is true	`8 GE 8 OR 5 GT 7 = YES`	4th

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: 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 month to first 2
limit district to 'BOSTON'
fetch sales gt 75000 labeled
```

This FETCH command returns the following data.

```
(MONTH JAN95, PRODUCT TENTS, DISTRICT BOSTON): FALSE
(MONTH FEB95, PRODUCT TENTS, DISTRICT BOSTON): FALSE
(MONTH JAN95, PRODUCT CANOES, DISTRICT BOSTON): FALSE
(MONTH FEB95, PRODUCT CANOES, DISTRICT BOSTON): TRUE
(MONTH JAN95, PRODUCT RACQUETS, DISTRICT BOSTON): FALSE
(MONTH FEB95, PRODUCT RACQUETS, DISTRICT BOSTON): FALSE
(MONTH JAN95, PRODUCT SPORTSWEAR, DISTRICT BOSTON): FALSE
(MONTH FEB95, PRODUCT SPORTSWEAR, DISTRICT BOSTON): FALSE
(MONTH JAN95, PRODUCT FOOTWEAR, DISTRICT BOSTON): TRUE
(MONTH FEB95, PRODUCT FOOTWEAR, DISTRICT BOSTON): TRUE
```

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.

Example: Controlling errors due to the setting of the DECIMALS option

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.

Example: Controlling errors when comparing floating point numbers resulting from arithmetic operations

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)
```

Example: Controlling errors when comparing SHORTDECIMAL and DECIMAL values

You cannot expect exact equality between SHORTDECIMAL and DECIMAL representations of a decimal number with a fractional component, because the DECIMAL data type has 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 a comparison of the SHORTDECIMAL number to a fractional decimal number is likely to return NO.

```
define sdvar shortdecimal
sdvar = 1.3
show sdvar eq 1.3
```

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.

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

- Do not mix the SHORTDECIMAL and DECIMAL types in comparisons. To avoid mixing these two data types, you should generally define variables with fractional decimal components as DECIMAL rather than 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 the textual value. Instead, it compares the positions of the values in the default status of the dimension. This allows you to specify commands like the following command.

```
fetch district lt 'SEATTLE' labeled
```

Commands are interpreted such as these using the process below.

1. The text literal `'SEATTLE'` is converted to its position in the DISTRICT dimension's default status list.
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 dimension's default status list and it returns NO for SEATTLE itself. It will also return NO for districts added after SEATTLE.

```
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 'JUN95' 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.

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 4-23. 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.

```
if bill.date + 60 le today
```

Dates also have a numeric value. For example, January 1, 1000, has a value of `-328717`; December 31, 9999, has a value of `2958464`; and January 1, 1900, has a value of `1`. Thus, each date in this range has a corresponding numeric value. For example, January 2, 1000, has a value of `-328716` and January 2, 1900, has a value of `2`. You can use the CONVERT function to change dates to integers and integers to dates and compare them.

Comparing dates and times

There are several types of time dimensions whose values are time periods. Each time period covers a range of dates, from one day to one year. If a date falls between the starting and ending dates of that time period, then is equal to a time period.

You can also compare one time dimension value to another time dimension value, when the two have the same length and phase. However, you cannot compare two time dimension values with the standard Boolean operators when they have different period lengths or phases. To make such a comparison, you can convert a time dimension value to a date (its value becomes the last day in the time period) and then compare it to another time period.

Correct: `show day lt convert(week date)`
Incorrect: `show day lt week`

The Boolean operator IN is designed for comparing time periods. It evaluates whether a date or time period is contained in another time period. The comparison is based on the ending dates of the time periods. If the ending date of the first period is in the second period, then the result is YES. It does not matter whether the first period is actually shorter or longer than the second.

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 ASCII value of the characters.

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.

Example: Comparing text data

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_ would match any text that contained 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 ('EXPRESS' like 'EX%')
```

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

What is a conditional expression?

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.

How is a conditional expression processed?

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: 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 'JAN96' to 'JUN96'
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   JAN96    FEB96    MAR96     APR96    MAY96    JUN96
--------- -------- -------- -------- ------- --------- ----------
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

What is a substitution expression?

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.

Note: You cannot use ampersand substitution in model equations.

Example: 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
```

How are substitution expressions processed?

Although ampersand substitution lets you write more 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.

Other ways to write general programs

To avoid ampersand substitution, you can use the IF or SWITCH command instead. For more information about the IF or SWITCH command, see the entry for the command in the OLAP DML Reference.

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
IF and SWITCH commands,	the entry for the command in the OLAP DML Reference
writing conditional commands in the OLAP DML,	“Controlling the Flow of Execution” on page 8-19

Working with NA Values

Definition: NA value

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.

When NA values are relevant

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.

Overview: 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 in the operation it is executing. 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 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, when an EXPORT (EIF file or pipeline) command is executed, the NATRIGGER property expression on a variable is not evaluated when the variable is simply exported; 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 kinds of expressions, not just functions or addition (+) and subtraction (-) operations, and also have 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.

Example 1: Using NAFILL

The following command uses NAFILL, but does not change the data in storage. It merely fetches the data in the SALES variable with each NA replaced with the number 1.

```
fetch nafill(sales, 1)
```

Example 2: Using NAFILL

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)
```

Example 3: Using NAFILL

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.

```
show average(nafill(sales 0.0) district)
```

5

Populating OLAP DML Data Objects

Chapter summary

This chapter provides an overview of how you populate OLAP DML data objects that hold source data and how to populate OLAP DML variables with calculated values.

List of topics

This chapter includes the following topics:

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

Overview: Populating an Analytic Workspace

Process: 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 members, are stored in dimension workspace objects.

Variables and dimensions can be populated:

- By loading data from the relational database. For example, you might load sales fact data into a variable from a sale fact table, load time dimension members from a time dimension table, custom dimension members from a customer dimension table, and product dimension members 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.
- As an alternative to loading data from the relational database, you can load data from a flat file using data loaders controlled through the OLAP DML.

There are other types of workspace objects that are discussed later in this guide. Like variables and dimensions, this objects must also be populated from the relational database, as the result of a calculation, or from a flat file.

Process: Populating data objects in an analytic workspace

To explicitly populate the source 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 the analytic workspace's 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 OLAP DML's SQL commands and data loading commands.

OLAP DML commands that populate source data objects

The OLAP DML commands that you typically use to populate source data objects are listed in the following table.

Command	Description
=	Assigns the results of an expression to a variable, option, or relation; or assigns the result of a model to a variable or, when the result is numeric, to a dimension value. For more information, see "Assigning Values to Data Objects" on page 5-13, "Definition: Solution variable" on page 7-2, and the topic for the EQUAL command in the OLAP DML Reference.
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 and the topic for the command in the OLAP DML Reference.
FILEREAD	Stores the data that is read from an input file into a dimension, composite, relation, or variable. For more information, see Chapter 11 and the topic for the command in the OLAP DML Reference.
SQL	Retrieves data from a relational database into a dimension or variable. For more information, see Chapter 10 and the topic for the command in the OLAP DML Reference.
IMPORT	Copies the data and definitions from one analytic workspace into another. For more information, see the topic for the command in the OLAP DML Reference.

Maintaining Dimensions and Composites

How do you specify dimension values?

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

You can add, delete, merge, reposition, or change dimension values using the MAINTAIN command. Consequently, storing and manipulating the values of a dimension is called maintaining the dimension.

Who can maintain dimensions?

You can only maintain a dimension when you have permission to both maintain and read the dimension. Maintain permission is implicitly denied whenever read permission is restricted for a dimension, even when you specify maintain permission for the dimension.

By default, you have permission to both read and maintain dimensions. However, either or both of these permissions can be changed using the PERMIT command.

For more information on using the PERMIT command, see "Adding Security to an Analytic Workspace" on page 2-20 and the topic for the PERMIT command in the OLAP DML Reference.

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's 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),	that dimension's pushed status list is cleared, and popping that dimension has no effect.

For more information on popping and pushing values, see "Introducing Dimension Status" on page 6-2 and the entries for the POP and PUSH commands in the OLAP DML Reference.

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 are dimensioned by the dimension you are maintaining 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 ...
```

Overview: 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 non-time dimension, a time dimension, or a composite.

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 MERGE with dimensions of any data type, including time data types. However, because the ADD keyword provides a quick way of adding time dimension values, the MERGE keyword may not be as useful with time dimensions as with TEXT or ID dimensions.

Adding values to non-time dimensions

You can use the MAINTAIN command with the ADD keywords to add values to a non-time 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: Adding values to non-time 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
```

Adding values to time dimensions

You can use the MAINTAIN command with the ADD keyword to add new values to time dimensions (that is, dimensions with the DAY, WEEK, MONTH, QUARTER, or YEAR data type). You can specify what values you want to add by specifying:

- The number of periods to add at the beginning or end of an existing list of dimension values.
- A list of values. (In this case, you can specify the values as text literals or as date or text expressions.)

Regardless of how you specify the values, keep the following points in mind:

- You can specify any date that falls within the time period you want to add. For example, to add the month January 1999, you can specify any date from 01JAN99 through 31JAN99. The DATEORDER option is used to resolve any ambiguities.
- When adding values to a time dimension that does not yet have values, you must specify only the first and last values you want to add for the dimension. The gaps are automatically filled in with appropriate values for the intervening time periods.

Adding date values to time dimensions

When you add a dimension value by specifying a DATE expression or a text value that represents a complete date, keep the following points in mind:

- If a time dimension already has values, then you can add values only at the beginning or the end of the existing list. To add values, you must specify only the first or last value you want to add. The gap between the existing list and the value you specify is automatically filled in.
- When you specify a time dimension value as a date, you must provide only the date components that are relevant for the type of dimension you are maintaining:
  - For a DAY or WEEK dimension, you must supply the day, month, and year components.
  - For a MONTH or QUARTER dimension, you must supply only the month and year (for example, JUN98 or 0698 for June 1998).
  - For a YEAR dimension, you must specify only the year (for example, 98 for 1998).

For more information on the valid input styles for dates, see the entry for the DATEORDER option in the OLAP DML Reference.

Adding text values to time dimensions

When you add a dimension value by specifying the values as text literals or TEXT or ID expressions (rather than as a date), keep the following points in mind:

- If you use a TEXT expression, then each element or line is treated as a separate value.
- The values can be in either one of the following formats:
  - The format specified by the VNF (value name format) for the dimension (or in the default format for the type of dimension you are maintaining when the dimension does not have a VNF). In this case, each value explicitly indicates the time period you want to add. For example, if the VNF for a MONTH dimension is `<MTXT><YY>`, then the value JAN99 represents the month January 1999.
  - A valid input style for date values. For more information on the valid input styles for dates, see the entry for the DATEORDER option in the OLAP DML Reference.

For more information on the default formats for time dimensions, see the entry for the VNF command in the OLAP DML Reference.

Example: Adding values to a time dimension

Suppose you define a new time dimension, called QTR, with a data type of QUARTER, and you add dimension values for the quarters in 1998 and 1999. You must add only the first and last dimension values you want, and the intervening values will be filled in automatically.

To add the first and last quarters, you can specify any dates that fall within those quarters.

```
define qtr dimension quarter
maintain qtr add '01JAN98' '31DEC99'
```

Displaying the default status list for the dimension shows the new dimension values.

```
Q1.98
Q2.98
Q3.98
Q4.98
Q1.99
Q2.99
Q3.99
```

Updating relations when you merge 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 the complete syntax for the MAINTAIN command, see the entry for the command in the OLAP DML Reference. For information about explicitly updating relations, see "Assigning Values to Data Objects" on page 5-13.

Example: Merging values into a composite

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. Using the MAINTAIN command with DELETE keyword, 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 it is sorted according to a specified criterion, the top or bottom *n* values of the dimension, or the top or bottom *n* performers, by percentage, of the dimension
- 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

You delete values from a dimension with a time data type (that is, DAY, WEEK, MONTH, QUARTER, or YEAR) the same way that you delete values from any other dimension except that you can delete values only from the beginning or the end of the existing list of values.

Example: 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.

Example: Deleting dimension values from a 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

For dimensions that have a non-time data type, 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.

Example: Changing the position of dimension values

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
   ```

   Note: 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
   ```

   Note: To sort the values alphabetically, sort by the dimension itself.

Note: You cannot use the MAINTAIN command to save the sorted order as the permanent order of a time dimension. The values of a time dimension must be stored in increasing chronological order.

For more information on using the SORT command, see the entry for the command in the OLAP DML Reference.

Example: Storing dimension values in alphabetical order

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. Conjoint dimensions are not automatically maintained. In programs, you use the MAINTAIN command to maintain the values in a conjoint dimension.

Assigning Values to Data Objects

Introducing the assignment statement

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.

Using OLAP DML objects in assignment statements

The following table outlines the OLAP DML 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 when the result of the expression is numeric	Yes
Composite	No	Yes
Worksheet	Yes	Yes
Function	No	Yes
Formula	No	Yes
Valueset	No	Yes

How values are assigned to variables

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 a loop is performed 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 1: Assigning values to variables

The demo analytic workspace contains the CHOICEDESC variable that is dimensioned by CHOICE. Before you enter data for the variable, the variable's cells 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 variable's cells 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
```

Example 2: Assigning values to variables

The following example shows an expression that is dimensioned by MONTH, PRODUCT, and DISTRICT and is assigned to a new variable. The expression calculates a 1997 sales plan based on unit sales in 1996.

```
define units.plan integer <month product district>
limit month to year 'YR97'
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 following happens 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: 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 dimension's default status list.

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. For more information on working with models, see Chapter 7 and the topic for the MODEL command in the OLAP DML Reference.

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 data object.

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 added to the variable.

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

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
how values are stored in data objects,	Chapter 3
assigning values to data objects,	the entry for the = command in the OLAP DML Reference
updating relations when you merge values into a dimension,	"Updating relations when you merge new values" on page 5-9
QDRs,	"Specifying a Single Value for the Dimension of an Expression" on page 4-16

Calculating and Analyzing Data

How to calculate and analyze data using the OLAP DML

Typically, using the OLAP DML, you calculate and analyze data in the following ways:

- Perform common calculations using built-in OLAP DML functions that are described in detail in the OLAP DML Reference and outlined in "Categories of OLAP DML functions" on page 5-19.
- Aggregate (or roll up) data in variables that are dimensioned by one or more hierarchical dimensions as outlined in "Aggregating Data" on page 5-19.
- Create populated solution variables using the MODEL object as described in Chapter 7.

Categories of OLAP DML functions

The OLAP DML provides built-in functions for numeric analysis. The categories of these functions are described below.

Category	Description
Numeric cell-by-cell	Functions that operate on each cell of an expression or variable.
Time-series	Functions that retrieve values from a previous or future time period and perform calculations on those values.
Forecasts and regression	Functions that analyze trends in your data.
Statistical	Functions that perform calculations for statistical analysis.
Financial	Functions that perform calculations for financial analysis.
Aggregation	Functions that return an aggregate value, generally consisting of a single value for many values of the input expression.

For more information on the functions in these categories, see the categorized list of functions in the OLAP DML Reference. For more information on working with numeric expressions, see "Numeric Expressions" on page 4-23.

Aggregating Data

What does "aggregating data" mean?

If you have a variable that is dimensioned by one or more hierarchical dimensions, then you can calculate the totals of the variable's data at the upper levels of each hierarchy from the detail data — that is, the data at the lowest level of the hierarchy. This is called *aggregating*.

For more information on hierarchical dimensions and variables defined with hierarchical dimensions, see "Defining Hierarchical Dimensions and Variables That Use Them" on page 3-20.

Specifying how and when data is aggregated

You can use the OLAP DML to aggregate the data. You define an aggregation map object that specifies which data should be pre-calculated when the AGGREGATE command is executed, and which data should be calculated on the fly.

For information about the aggregation map (AGGMAP) object, the AGGREGATE command and function, and other aggregate commands (such as RELATION, CACHE, and AGGINFO), see the OLAP DML Reference. For information on designing, writing, and debugging programs, see Chapter 8 and Chapter 9.

6

Limiting an Application's View of the Data

Chapter summary

This chapter introduces dimension status and the use of the LIMIT command to temporarily change your view of the data in an analytic workspace.

List of topics

This chapter includes the following topics:

- Introducing Dimension Status
- Limiting Using a Simple List of Values
- Limiting Using a Boolean Expression
- Limiting to the Top or Bottom Values of a Sorted Dimension
- 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
- Working with Null Status
- Working with Valuesets

Introducing Dimension Status

Definition: Current status list

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.”

If you are familiar with relational database terminology, then you can think of the current status list of a dimension as a view of the dimension. Whether or not a dimension value is in status determines your view of the data from all of the objects that are dimensioned by it. In general, when an OLAP DML command is processed, only those values that are in status are accessed.

- For dimensions, only those dimension values that are in the current status list are accessed; and for dimensioned objects, only those values that are indexed by dimension values that are in the current status list are accessed.
- As a loop is performed through dimensioned objects, it uses the order of the dimension values in the current status list to determine the order in which to access the object’s values.

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.

Definition: Default status list

When you first attach an analytic workspace, the current status list of each dimension consists of all of the dimension’s values 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.

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. For more information on storing and maintaining dimension values, see “Maintaining Dimensions and Composites” on page 5-3 and the entry for the MAINTAIN command in OLAP DML Reference.
- You can change the read permission of values that are associated with a dimension by using the PERMIT command or the PERMITRESET command.

For more information on using these commands, see "Adding Security to an Analytic Workspace" on page 2-20 and the topics for the commands in OLAP DML Reference.

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 a dimension's current status list.
- The SORT command to arrange the order of values in a dimension's current status list.

If you are familiar with relational database concepts and terminology, then it may help you to think of using a LIMIT command to "set or change dimension status" or "limit a dimension" in an analytic workspace as similar to using the SQL SELECT statement to "select a view" in a relational database. Changing dimension status merely restricts your view of the data during a given session. No matter how you change the current status list of a dimension, the changes have no permanent effect on your analytic workspace; every dimension retains all of its values.

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.

Command or function	Description
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 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 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, an option, a single-cell variable, a valueset, 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 and update the saved object values, whereas 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's execution.

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
storing and maintaining values in a dimension,	"Maintaining Dimensions and Composites" on page 5-3
the CONTEXT, POP, POPLEVEL, PUSH, and PUSHLEVEL commands,	"Preserving the Session Environment" on page 8-25 the entries for the commands in OLAP DML Reference
changing the read permission of dimension values using the PERMIT or PERMITRESET commands,	"Adding Security to an Analytic Workspace" on page 2-20 the topics for the commands in OLAP DML Reference
sorting dimension values,	the entry for the SORT command in OLAP DML Reference

Limiting Using a Simple List of Values

Overview: 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. 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.

Example: Limiting to literal values

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
```

Limiting using time values

You can use the LIMIT command to limit dimension status for the value of a time dimension. When you specify a value of a time dimension (that is, a dimension with a data type of DAY, WEEK, MONTH, QUARTER, or YEAR), the value can be in:

- The format specified by the VNF (value name format) for the dimension (or in the default VNF for the type of dimension you are limiting when the dimension does not have a VNF).
- A valid input style for date values, as described in the DATEORDER option.

When you specify a time dimension value as a date, you only need to provide the date components that are relevant for the type of dimension you are limiting.

IF you specify a value for a . . .	THEN you must specify the . . .
DAY or WEEK dimension,	day, month, and year.
MONTH or QUARTER dimension,	month and year (for example, JUN95 or 0695 for June 1995).
YEAR dimension,	year (for example, 95 for 1995).

If you specify a DATE expression or a text value that represents a complete date, then you can specify any date that falls within the time period that is represented by

the desired dimension value. The DATEORDER option is used automatically to resolve any ambiguities.

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
limiting dimensions,	the rest of this chapter the entry for the LIMIT command in OLAP DML Reference
the valid input styles for dates,	the entry for the DATEORDER option in OLAP DML Reference
VNF (value name format),	the entry for the VNF command in OLAP DML Reference

Limiting Using a Boolean Expression

Overview: 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.

Example: Limiting using a Boolean expression

In this 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 to construct a Boolean expression

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
  ```

How LIMIT handles Boolean expressions with more than one dimension

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's 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.

Example: How LIMIT handles Boolean expressions with many dimensions

Suppose that the MONTH, DISTRICT, and PRODUCT dimensions of the demo analytic workspace 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 `sales gt 90000` (as shown below) would give you your desired result.

```
limit product to sales gt 90000
status product
report sales
```

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's 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.

```
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 a dimension to all dimension values that match the 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: Limiting using the ANY function with a Boolean expression

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)
status product
report sales
```

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.

```
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
```

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
limiting dimensions,	the rest of this chapter the entry for the LIMIT command in OLAP DML Reference
creating expressions,	Chapter 4
converting data types,	the entry for the CONVERT function in OLAP DML Reference
the ANY function,	the entry for the function in OLAP DML Reference

Limiting to the Top or Bottom Values of a Sorted Dimension

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
```

Example: Limiting to the top or bottom values

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
```

Limiting to the top or bottom performers, by percentage

You can 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: Limiting to the top or bottom performers by percentage

Suppose you want to sort products in descending order by each product's contribution 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 1995 of products in the Boston district that reached or exceeded 30 percent of all sales.

```
limit month to 'JAN95' 'FEB95' 'MAR95'
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           JAN95       FEB95       MAR95
--------------- ---------- ---------- ----------
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

Overview: Limiting to the values of related dimensions

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 the related-dimension values. If valuelist is omitted, then the current status of related-dimension is used.

For the complete syntax for the LIMIT command, see the entry for the command in OLAP DML Reference.

Example: Limiting with 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 status is determined when you limit to a related dimension

When you limit a dimension to a related dimension, the current status list is created in a two-step process, as shown in the following table.

1. The values in the dimension's 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's current status list are arranged in the order of their default status list.

Suppressing the sort when you limit 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 using related time dimensions

Every time dimension (with a data type of DAY, WEEK, MONTH, QUARTER, or YEAR) is related to every other time dimension through an implicit relation. When you limit the values of a time dimension by specifying another time dimension as the related dimension, the implicit relation is used by default.

For example, you can issue the following command.

```
limit month to quarter year
```

This command will temporarily limit QUARTER to YEAR, then limit MONTH to QUARTER, and finally restore QUARTER to its original status.

However, if an explicit relation is defined between the two time dimensions, then you can override the default by specifying the name of the explicit relation as the related dimension.

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
limiting dimensions,	the rest of this chapter the entry for the LIMIT command in OLAP DML Reference
working with values in time dimensions,	"Defining Dimensions" on page 3-4 "OLAP DML Data Types" on page 4-2
sorting the current status list,	the entry for the SORT command in OLAP DML Reference

Limiting Based on the Position of a Value in a Dimension

Overview: Limiting based on a value's position 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 a value's 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 dimension's full set of 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.

For the complete syntax for the LIMIT command, see the entry for the command in OLAP DML Reference.

Limiting using a value's position in an unrelated dimension

You can use the LIMIT command with the NOCONVERT keyword to insert a value into a dimension's 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 to keyword to insert a value into a dimension's status list based on the numeric position of the values in the status list of the unrelated dimension 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.

For the complete syntax for the LIMIT command, see the entry for the command in OLAP DML Reference.

Limiting Based on a Relationship Within a Hierarchy

Overview: Limiting based on a relationship within a hierarchy

You can use the LIMIT command to 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.

Syntax: Limiting based on a relationship within a hierarchy

The simplified syntax for using the LIMIT command to limit 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.

For more information on using the HIERARCHY keyword, see "Differences between HIERARCHY and DESCENDANTS keywords" on page 6-17. For the complete syntax of the LIMIT command, see the entry for the command in OLAP DML Reference.

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: 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: 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 example, when you use KEEP or REMOVE instead of TO with the LIMIT command, the SORT command has no effect.

Example: 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.

Step 1

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
```

Step 2

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
```

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
limiting dimensions,	the rest of this chapter the entry for the LIMIT command in OLAP DML Reference
hierarchical dimensions	“Defining Hierarchical Dimensions and Variables That Use Them” on page 3-20

Limiting Composites and Conjoint Dimensions

How to limit a composite

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.

Example: 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'
status coupons
The current status of MONTH is:
ALL
The current status of PRODUCT is:
SPORTSWEAR
The current status of MARKET is:
ALL
```

Ways of limiting a conjoint dimension

You can limit a conjoint dimension in either of the following ways:

- Limit the base dimensions.
- Limit the conjoint dimension itself.

Limiting a conjoint dimension 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'
```

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
limiting dimensions,	the rest of this chapter the entry for the LIMIT command in OLAP DML Reference
conjoint dimensions,	"Defining Variables That Handle Sparse Data Efficiently" on page 3-15

Working with Null Status

Setting the current status to null (empty 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.

Note: 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's 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 you limit 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

Definition: Valueset

A valueset is an OLAP DML 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 Services 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: Creating a valueset

This example adds the valueset LINESET to the demonstration analytic workspace. It is dimensioned by LINE and, therefore, it can be limited by the current values of the LINE dimension. The LD command attaches a description to the object.

The following OLAP DML commands produce the output shown below them.

```
limit line to first 2
status line
The current status of LINE is:
REVENUE, COGS
```

The following OLAP DML commands produce the output shown below them.

```
define lineset valueset line
ld Valueset for LINE dimension values
limit lineset to line
show values(lineset)
REVENUE
COGS
```

Limiting using a valueset

Once you have defined a valueset, you can use it to limit a dimension with a single LIMIT command.

For example, the following command limits the LINE dimension to the values stored in the LINESET valueset and displays the new status of LINE.

```
limit line to lineset
status line
The current status of LINE is:
REVENUE, COGS
```

Example: 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 following OLAP DML command produce the output shown below it.

```
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.

For the complete syntax of the LIMIT command, see the entry for the command in OLAP DML Reference.

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.

Command or function	Description
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.

For more information on these commands and functions, see the entry for the command or function in OLAP DML Reference.

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.

7

Working with Models

Chapter summary

This chapter describes how to use OLAP DML models to calculate data.

List of topics

This chapter 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

Definition: OLAP DML model

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.

Definition: Solution variable

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: Creating an OLAP DML 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 OLAP Worksheet or the MODEL command to enter 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 enter the lines shown in the following program.

```
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 enter 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 will be 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 dimension's initial status.

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.

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
overall understanding of the modeling capabilities of the OLAP DML,	the entry for the MODEL command in OLAP DML Reference
individual OLAP DML commands,	the entry for the command in OLAP DML Reference

Creating a Nested Hierarchy of Models

How to include one model in another

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.

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
overall understanding of the modeling capabilities of the OLAP DML,	the entry for the MODEL command in OLAP DML Reference
about masked equations and what-if analyses,	the entry for the INCLUDE command in OLAP DML Reference
individual OLAP DML commands,	the entry for the command in OLAP DML Reference

Basic Modeling Commands

OLAP DML commands for defining and running models

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	Enters completely new contents into 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 dictionary. 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.
- If a model equation is based on a time dimension (with a data type of DAY, WEEK, MONTH, QUARTER, or YEAR), then you must use the dimension's VNF (value name format), rather than a date format, to specify the dimension's values.
- 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 model's target dimension 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 7-8 explains how the name references are resolved.

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
overall understanding of the modeling capabilities of the OLAP DML,	the entry for the MODEL command in OLAP DML Reference
assigning values to objects,	"Assigning Values to Data Objects" on page 5-13 the entry for the = command in OLAP DML Reference
individual OLAP DML commands,	the entry for the command in OLAP DML Reference

Compiling a Model

Using the COMPILE command

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.

Resolving name references

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.

Analyzing dependencies with equation blocks

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.

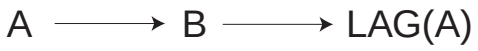
If a block contains equations that solve for values A, B, and C, then a noncyclic dependence can be illustrated as shown below where the arrows indicate that A depends on B, and B depends on C.

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.

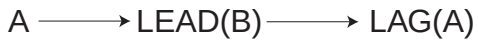
Dimensional dependence typically occurs over a time dimension. For example, it is common for a line item value to depend on the value of the same line item or a different line item in a previous time period. If a block contains equations that solve for values A and B, then a one-way dimensional dependence 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.



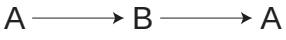
Simultaneous blocks

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.



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.



Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
overall understanding of the modeling capabilities of the OLAP DML,	the entry for the MODEL command in OLAP DML Reference
individual OLAP DML commands,	the entry for the command in OLAP DML Reference

Running a Model

Points to remember when 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 7-12.
- 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 OLAP DML Reference 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. For more information about the options, see the entry for the MODEL command in OLAP DML Reference.

Debugging a Model

How do you debug a model?

You debug a model in much the same way that you debug an OLAP DML program. There are two main methods for debugging OLAP DML programs. As outlined below, the method that you use depends on the degree of debugging that you want to perform.

Method	Description
Debugging file	Creates a debugging file that logs the progress of a program execution so you can analyze it for errors.
OLAP DML debugger	Allows you to interactively step through programs one line at a time and displays the current values of OLAP DML objects. The OLAP DML debugger is used from within OLAP Worksheet.

For more information on debugging in the OLAP DML, see Chapter 9.

Tools for debugging models

The OLAP DML provides an assortment of tools that will help you debug your models. You use these tools in OLAP Worksheet. These tools are listed in the following table.

Tool	Purpose
MODTRACE	An option that controls whether each line of a model is displayed while you run the model. When MODTRACE is set to YES, the model lines are displayed, and you can observe the order in which the equations are solved.
TRACE	A command that lets you step through the model line by line or block by block. At each step, model execution is suspended so that you can type special debugger commands or any other OLAP DML commands to examine the model environment. The debugging environment and the TRACE command are available only when you are using OLAP Worksheet.
WATCH	A command that lets you monitor the value of specific assignment targets in a model. Each time the target is assigned a new value, this value is displayed. The debugging environment and the WATCH command are available only when you are using OLAP Worksheet.
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.

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
the programs that produce debugging reports,	the entry for the MODEL command in the OLAP DML Reference
using the TRACE and WATCH commands,	the entry for the command in OLAP DML Reference Chapter 9

Modeling for Multiple Scenarios

Calculating several sets of figures

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, you 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 next few sections.

Defining a scenario dimension

You can call the scenario dimension SCENARIO, and give it values that represent the scenarios you want to calculate. For this example you can give it the values OPTIMISTIC, PESSIMISTIC and BESTGUESS.

```
define scenario dimension text
ld Names of scenarios
maintain scenario add optimistic pessimistic bestguess
```

Defining a solution variable dimensioned by the scenario dimension

For this example the solution variable should be dimensioned by DIVISION as well as by SCENARIO. Like the BUDGET variable in the demo analytic workspace, your solution variable can also be dimensioned by MONTH and by LINE. You can call the variable PLAN.

```
define plan decimal <month line division scenario>
ld Scenarios for financials
```

Entering input data into the solution variable

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')
```

Writing a model to calculate results based on the input data

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.

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
overall understanding of the modeling capabilities of the OLAP DML,	the entry for the MODEL command in OLAP DML Reference
individual OLAP DML commands,	the entry for the command in OLAP DML Reference

8

Designing Programs

Chapter summary

This chapter provides information about writing, compiling, testing, and calling programs that are written in the OLAP DML.

List of topics

This chapter includes the following topics:

- Introduction to OLAP DML Programs
- Invoking 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 Programs

Introduction to OLAP DML Programs

Definition: OLAP DML program

An OLAP DML program is a stored procedure, which is written in the OLAP DML, that acts on data in the analytic workspace and helps you accomplish some analytic workspace management or analysis task. You can write OLAP DML programs to perform analytic workspace tasks that you must do repeatedly, or you can write them as part of an application that you are developing.

Types of programs

Two main types of OLAP DML programs

There are two main types of OLAP DML programs: programs that do *not* return values and programs that return values.

How you can use programs that do not return values

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.

How you can use programs that return values

You can use a user-defined function in commands and expressions in the same way that you use built-in OLAP DML functions. For more information on user-defined functions, see "Writing User-Defined Functions" on page 8-16. For more information on built-in OLAP DML functions, see the OLAP DML Reference.

Related information

For more information about invoking, writing, and compiling programs, see the rest of this chapter; for more information on testing and debugging OLAP DML programs, see Chapter 9.

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.

For more information on these tasks, see the following table.

IF you want documentation about . . .	THEN see . . .
using data from a SQL database,	■ Chapter 10 ■ the entry for the SQL command in OLAP DML Reference
reading data from a file,	Chapter 11.
producing reports,	Chapter 12.
using a model for financial data,	■ Chapter 7 ■ the entry for the DEFINE MODEL command in OLAP DML Reference
details on the syntax and usage of individual OLAP DML commands, functions, options, and programs,	OLAP DML Reference.

Invoking Programs

Invoking programs that do not return values

There are two ways that you can invoke an OLAP DML program that does not return a value:

- Using the CALL command — You can invoke a program 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 HELLO that takes a text literal as an input argument. You can use the CALL command in the main program of your application to invoke the program.

  ```
  call hello ('Hello World')
  ```

  You typically use the CALL command to invoke a program when you are using an OLAP DML program that does not return values as a subprogram.

- As a command — You can invoke a program as a command. In this case, you do not enclose the program's arguments in parentheses, and the arguments are passed as text strings (not by value). For example, you can invoke the HELLO program in OLAP Worksheet by issuing the following command.

  ```
  hello 'Hello World'
  ```

You typically invoke a program in this way when it is a standalone or main program.

Syntax: CALL command

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* and *arg2* arguments are optional and specify any arguments that are expected by the called program. You can declare these arguments in the called program with the ARGUMENT command, or you can reference them in the program with the ARG function. When the program uses the ARGUMENT command and you use the CALL command to invoke the program, specify the arguments so that they match the positions of the arguments that are declared in the called program.

For the complete syntax of the CALL command, see the entry for the command in OLAP DML Reference.

Invoking user-defined functions

A user-defined function is a program that does not return a value. You invoke user-defined functions in the same way as you use OLAP DML built-in functions. You merely use the program's name in an expression and enclose the program's arguments, if any, in parentheses. The arguments are passed by value, not as text.

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 as standalone programs or invoke them using the CALL command, in these cases, the return value of the function is discarded.

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
using arguments,	"Passing Arguments" on page 8-11
the CALL command,	the entry for the command in OLAP DML Reference
user-defined functions,	"Writing User-Defined Functions" on page 8-16

Defining and Editing Programs

Defining a program

A program, like a dimension or a variable, is an OLAP DML object. You can define it using the DEFINE command. The following example defines a program named HELLO using the DEFINE command.

```
define hello program
```

Once you have defined a program object, you need to add command lines to it using an editor.

Editing programs

OLAP DML programs can be editing using a program editor or using commands in the OLAP DML.

Using an editor

OLAP Worksheet provides an editor that you can use to edit programs and formulas. To access the program editor from within OLAP Worksheet, type the EDIT command followed by the program name.

To save the program, choose **Save** from the File menu in OLAP Worksheet.

Procedure: Using OLAP Worksheet to edit a program

If you want to edit the program using OLAP Worksheet, then follow the procedure outlined below.

1. Start OLAP Worksheet.
2. Connect to an analytic workspace.

   Refer to the OLAP Worksheet Help system for details.
3. If necessary, define the program.

   For example, to define a program named SALESREP, enter the following command in the Command Input window in OLAP Worksheet.

   ```
   define salesrep program
   ```
4. To use the Edit Window in OLAP Worksheet, enter the EDIT command and the program name in the Command Input window.

   For example, to edit the SALESREP program, enter the following command.

   ```
   edit salesrep
   ```

   If you do not want to use the Edit Window, you can enter the program contents in the Command Input window. You can use one of the following modes.

   - Enter one line of code at a time in the Command Input window. For example, enter PROGRAM, enter each line of code, and then enter END to stop adding contents to the program. (Default mode)
   - Choose **Preferences** from the Options menu in OLAP Worksheet. In the Preferences dialog box, deselect **Execute on Enter.** You can now enter the entire contents of the program in the Command Input window. To save the program contents, choose the play button or Ctrl-Enter on the keyboard.

Using OLAP DML commands

The OLAP DML allows you to edit the contents of a program from the OLAP Worksheet command line or using an OLAP DML program. You may edit the contents of a program immediately after it has been defined, or immediately after using the CONSIDER command.

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 characters.
- 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.

  Note: 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 semi-colons (;).
- 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.

Example: Defining and add contents to a simple program

The following program named HELLO produces the phrase “Hello World.”

```
DEFINE HELLO PROGRAM
PROGRAM
show 'Hello World'
END
```

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
escape sequences,	“Text data types” on page 4-3
individual OLAP DML commands,	the topic for the command in OLAP DML Reference

Using Variables in Programs

Temporary and local OLAP DML variables

Variables, such as SALES or UNITS, that hold the data in your analytic workspaces are permanent variables. These variables persist from one OLAP Services 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 OLAP Services session. When you update the analytic workspace, only the definitions of the variables are saved. When you exit from the analytic workspace, the data values are discarded.
- A local variable is a single-cell variable that exists only for the duration of the program in which it is defined. 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 you should 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 and inplace 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. This approach forces you to write modular programs with clear input and output responsibilities.

Most applications combine these approaches, using permanent and inplace 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.

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 OLAP DML objects (for example, the OBJ function), the defined variable takes precedence.

The *datatype* argument specifies the data type of the local variable. A local variable can have a data type of BOOLEAN, DATE, DECIMAL, ID, INTEGER, SHORTDECIMAL, SHORTINTEGER, or TEXT.

For the complete syntax of the VARIABLE command and for a list of the commands and functions for which the defined variable takes precedence, see the entry for the VARIABLE command in OLAP DML Reference. For more information on data types, see "OLAP DML Data Types" on page 4-2.

Example: Defining local variables

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
   .
   .
   .
```

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
OLAP DML data types,	“OLAP DML Data Types” on page 4-2
permanent variables,	The topic for the DEFINE VARIABLE command in OLAP DML Reference
temporary variables,	The topic for the DEFINE VARIABLE command in OLAP DML Reference
local variables,	the topic for the VARIABLE command in OLAP DML Reference
individual OLAP DML commands,	the topic for the command in OLAP DML Reference

Passing Arguments

Two methods for accepting 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 allows you to use both simple and complex arguments, such as expressions. ARGUMENT 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. For information on these functions, see OLAP DML Reference.

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: 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')
```

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 1: Using 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)
```

Example 2: Using multiple arguments

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 OLAP DML 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.

Example: 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.

Example: 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.tbl '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
```

Ampersand substitution and performance

It is not possible to 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.

Passing OLAP DML object names and keywords

For the following types of arguments, you must always use an ampersand to make the appropriate substitution:

- Names of OLAP DML 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: Passing OLAP DML 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
```

Passing expression arguments by value

You can also pass expressions into a program by value. This means that the program receives its argument as the values that are the result of an expression rather than as the expression itself. Because this type of argument provides only the value of the expression, it does not give the program access to any OLAP DML object that is used in the original expression.

To pass an expression's value as an argument, you must execute the program with the CALL command and enclose its arguments in parentheses immediately following the program name.

```
CALL programname(argument1 [argument2 ...])
```

By using the CALL command to execute a program from within another program, you can use expressions to construct arguments "on the fly." In this way, the arguments passed to the second program can vary according to what has already happened in the first program.

Example 1: Passing expression arguments by values

Suppose the first argument for your program PRODUCT.RPT specifies the text that you want used for NA values. You might already have that text stored in a variable (for example, TEMP1). Rather than supplying the text as an argument, you can specify the name of the variable as an argument.

```
call product.rpt(temp1)
```

In this case, the argument that is passed into the PRODUCT.RPT program is not the literal text `'temp1'`, but the current value of the variable TEMP1. You can still pass literal text into the program, but you must enclose the text in single quotation marks.

```
call product.rpt('Missing')
```

Example 2: Passing expression arguments by values

In the program lines below, an argument has been passed that indicates the type of report that should be produced. The report program is called with an argument that specifies the text to be used for NA values. Because that text varies with the type of report, an expression is used as the argument to supply the appropriate text.

```
argument reptype text
call product.rpt(if reptype eq 'Revenue' then 'Missing' else 'Not Available')
```

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
ampersand substitution,	"Substitution Expressions" on page 4-39
individual OLAP DML commands,	the topic for the command in OLAP DML Reference

Writing User-Defined Functions

Definition: User-defined function

When an OLAP DML program returns a value, it is called a user-defined function. This means you can use it in commands and expressions in the same way you use OLAP DML functions.

A user-defined function is a program that contains a RETURN command followed by an expression. The RETURN command returns a single value when the program terminates.

```
RETURN expression
```

Data type of a user-defined function

When you create a user-defined function, 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 context from which the program was called.

For the complete syntax of the DEFINE PROGRAM command, see the entry for the command in OLAP DML Reference.

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 when you are using the function in a context that loops over a dimension (for example, in a REPORT command), then the function returns results with the same dimension 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. For more information about using arguments with programs, see "Passing Arguments" on page 8-11 and the entry for the ARGUMENT command in OLAP DML Reference.

Example: User-defined function

Description

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.

Program Code

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
```

Invoking the Program

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)
```

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
invoking programs that return values,	"Invoking Programs" on page 8-3
using arguments with programs,	"Passing Arguments" on page 8-11 the entry for the ARGUMENT command in OLAP DML Reference
built-in OLAP DML functions,	the OLAP DML Reference
individual OLAP DML commands,	the topic for the command in OLAP DML Reference

Controlling the Flow of Execution

Control structures that modify the sequence of command execution

Ordinarily, the lines of a program are executed sequentially — in linear fashion. However, a well-designed program controls the flow of execution by using OLAP DML 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.

Command or Keyword	Action	Event that Triggers Action
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.

For more information on an individual command, see the entry for the command in OLAP DML Reference.

Guidelines for constructing a label

When creating a label, follow these guidelines:

- The first character in the label must be a letter, a period (.), or an 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 characters are unique. A label can contain up to 3999 characters (the maximum length of a text line minus 1 character for the colon that identifies a label). However, because only the first eight characters of a label name are used, you can experience problems with label names greater than eight characters when the first eight characters 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: Using the FOR command to loop over the values in a given dimension

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
```

Example: Using the FOR command to loop over the values in several dimensions

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: Using the FOR command to loop over values in several dimensions

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 specified DISTRICT in the FOR loop, but not PRODUCT, then all the values of PRODUCT would be in status while the FOR loop executed. The IF command would then test data for only the first value of the PRODUCT dimension.

Examples: Branching in a program to avoid setting null status

Example: Branching using IFNONE keyword

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: Alternative to branching using IFNONE keyword

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

Directing output to a file

To send output to a file, use the OUTFILE command followed by a file name. A file will be created with that name. 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 you route 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.

Example: 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 the working directory of your analytic workspace. You can specify a full path name when you want to use a different drive or directory.

```
outfile 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 yearend.txt
year.end.expenses
outfile eof
```

Routing error messages

You can route error messages to a file 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.

The next topic explains that you can use the DBGOUTFILE command to create a log, or debugging file, of your program's execution. When you create a debugging file and set ECHOPROMPT to YES, input lines and error messages are routed to the debugging file instead of to the current outfile.

If you set ECHOPROMPT to YES, then remember to save and restore its original value with the PUSH and POP commands.

Setting paging options

Paging options such as BMARGIN and LSIZE have separate values for the default outfile and for files. Executing the OUTFILE command sets the paging options to their current values for the specified output destination. To make sure the paging options have the values you want, set them after executing the OUTFILE command.

When you set paging options for the default outfile, the new values remain in effect until you reset them. However, when you set paging options for a file, the new

values remain in effect only as long as you continue sending output to the same file. When an OUTFILE command that routes output to a different destination is executed, including a different file, the paging options return to their default values for files.

Therefore, if you want the paging options to have a particular value for a file, then you must reset the options each time you use the OUTFILE command for the file.

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
creating a debugging file and debugging with files,	“Debugging with a Debugging File” on page 9-2
individual OLAP DML commands,	the topic for the command in OLAP DML Reference

Preserving the Session Environment

Environment settings

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 the program and session 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, an option, a single-cell variable, a valueset, 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 of OLAP Services. With contexts, you can access and update the saved object values, whereas 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.

Using PUSH to save a dimension’s status or an option’s value

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.

Using PUSH to save 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 PUSHLEVEL and POPLEVEL to save several values at once

If you are saving the values of several dimensions and options, then the PUSHLEVEL and POPLEVEL commands provide an alternative and more 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.

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'
```

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.

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 and CONTEXT function. With these, you can access and update your saved object values, as well as save and restore them. For details about using named contexts, see the entries for the CONTEXT command and the CONTEXT function in OLAP DML Reference.

Handling Errors

Overview: 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, the following actions are performed.

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.

To find out what the value of ERRORNAME will be for specific error conditions, you can check the dimension _MSGID in the `express.db` analytic workspace. The error messages are contained in the variable _MSGTEXT, which is dimensioned by _MSGID. To see this list, execute the following command.

```
report w 60 _msgtext
```

Many of the error messages contained in _MSGTEXT are constructed so that appropriate values can be substituted in the message at the time it is produced (for example, the name of an OLAP DML object). These substitutions are indicated by a percent sign (%) followed by one or more characters in the _MSGTEXT value. In most cases, you can understand the purpose and condition of the message without knowing exactly what will be substituted.

When you need to, you can use the SIGNAL command to send to the current outfile the ERRORNAME and ERRORTEXT of the last error that occurred. The SIGNAL command has the following format.

```
SIGNAL errorname [message]
```

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: 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 do not specify your own message in a SIGNAL command, then Express the error name and a default message are produced.

```
ERROR: (TOOMANY) Please contact the administrator of your Oracle Express Server
application.
```

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:

- Method 1 — The error message is produced immediately, and the error condition is then passed through the chain of programs.
- Method 2 — 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: Producing the error message immediately

For Method 1, 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: Producing the error message at the end of the chain

For Method 2, 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
```

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
testing and debugging programs,	Chapter 9
individual OLAP DML commands,	the topic for the command in OLAP DML Reference

Compiling Programs

When a program is compiled

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 after you have edited it.

Example: Using the COMPILE command

The following is an example of a COMPILE command that compiles the MYPROG program.

```
compile myprog
```

For the syntax of the COMPILE commands, see the entry for the command in OLAP DML Reference.

Example: Compilation errors

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: MOTNH does not exist in any attached database.
In MYPROG PROGRAM:
limit motnh to last 6
     ^
```

You can edit the program to correct the error and then try to compile it again.

How a program is compiled

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.

Note: Program lines that include ampersand substitution will not be compiled. However, the presence of such lines does not constitute an error. A program whose other lines compiled correctly is considered to be a compiled program.

Compiling and updating

After you compile a program, the compiled code is used each time you run the program in the current OLAP Services session. When you update your analytic workspace after compiling a program, the compiled code is saved in your analytic workspace and uses it when you run the program in future sessions. Therefore, you should be sure to update your analytic workspace after compiling a program.

Compiling, exporting, and importing

After you export a program from a source analytic workspace, if you import the program to a target analytic workspace, then the compiled code is not exported or imported. Therefore, after you import a program to a target analytic workspace, you need to compile the program and update the target analytic workspace.

To keep an application analytic workspace compact and uncluttered, application builders often define and test objects in a test analytic workspace, and then import the tested objects to the final analytic workspace. When you follow this procedure, remember to compile your programs in the final analytic workspace.

Compiling with object definitions

When your program defines an object and then uses the object in the program, the program will not compile. When COMPILE encounters the reference to the object, it treats the reference as a misspelling because the object does not yet exist in the analytic workspace.

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')
```

For the syntax of the OBJ function, see the entry for the function in OLAP DML Reference.

Testing Programs

Testing a program by running it

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

Using SHOW commands

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 next section.

For the syntax of the SHOW command, see the entry for the command in OLAP DML Reference.

Using the BADLINE option

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.

For the syntax of the BADLINE option, see the entry for the option in OLAP DML Reference.

Example: 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.  (If you want NA to be
returned as the result of a division by zero, set the DIVIDEBYZERO option to
YES.)
In TEST PROGRAM:
myint2 = 250/myint1
```

9

Debugging Programs

Chapter summary

This chapter explains how to debug programs that are written in the OLAP DML.

List of topics

This chapter includes the following topics:

- Overview: Debugging in OLAP DML
- Debugging with a Debugging File
- Debugging with OLAP Worksheet
- OLAP DML Debugger Commands

Overview: Debugging in OLAP DML

How you debug OLAP DML

There are two main methods for debugging OLAP DML programs: using a debugging file, and using the OLAP DML debugger from within OLAP Worksheet. The method that you use depends on the degree of debugging that you want to perform and how the OLAP DML code is executed.

Debugging using a debugging file

If you are executing OLAP DML code through the OLAP API, you can use a debugging file that logs the progress of program execution so you can analyze it for errors.

For more information on working with a debugging file, see “Debugging with a Debugging File” on page 9-2.

Debugging using the OLAP DML debugger

If you are executing OLAP DML code from within OLAP Worksheet, you can use the OLAP DML debugger when you want to debug by interactively stepping through a program one line at a time while displaying the current values of OLAP DML objects.

The exact steps that you take to use OLAP Worksheet depend on the location of the program you want to debug and the type of session in which OLAP Worksheet runs. For more information, see “Debugging with OLAP Worksheet” on page 9-5, and “OLAP DML Debugger Commands” on page 9-6.

Debugging with a Debugging File

Why you debug using 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

Command you use to create a debugging file

To create a debugging file, you use the DBGOUTFILE command. The simplified syntax of the DBGOUTFILE command is shown below.

```
DBGOUTFILE {EOF|[APPEND] file-id [NOCACHE]}
```

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 argument *file-id* 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 the complete syntax of the DBGOUTFILE command, see the entry for the command in OLAP DML Reference, which you can access by selecting **Language** from the Help menu in OLAP Worksheet.

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.

As outlined below, using either the ECHOPROMPT or IFCOPY option, you can also specify that additional information should be included in the debugging file.

IF you want the debugging file to interweave the program lines with . . .	**THEN set the . . .**
both the program's input and error messages,	ECHOPROMPT option to YES.
only the program's input,	IFCOPY option to YES.

For the syntax of the ECHOPROMPT, IFCOPY, and PRGTRACE options, see the entry for each option in OLAP DML Reference.

Example: Debugging using a debugging file

Creating a debugging file

The following commands create a useful debugging file called `debug.txt` in the current working directory.

```
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
```

Sample Program Code

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
```

Debugging File Output

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 should be 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.
```

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
specifying the file to use for output,	“Directing Output” on page 8-23
individual OLAP DML commands,	the entry for the command in OLAP DML Reference

Debugging with OLAP Worksheet

How does OLAP Worksheet help you debug?

OLAP Worksheet allows you to use the TRACE or WATCH commands to interactively step through your program, pausing to examine the current values of OLAP DML objects and specifying how many program lines to execute. It also allows you to set watch points where execution will stop when data values meet certain conditions.

Starting the OLAP DML debugger

Access to the OLAP DML debugger is through the TRACE command, which controls the list of programs that are traced by the debugger. To use the TRACE command, you must be connected to OLAP Services using OLAP Worksheet.

To debug a single program, attach the analytic workspace in which the program resides, and use the TRACE command to add that program to the trace list.

```
execute 'trace quarter.rpt'
```

Procedure: Debugging a program located on the OLAP Services computer

To debug a program that is located on the same computer as that on which OLAP Services is running, take the following steps:

1. Choose **Connect** from the File menu to display the Login dialog box.
2. Specify the server name, password, and computer name in the Login dialog box. If necessary, enter a user name, domain name, and password in the appropriate boxes. For details, refer to the OLAP Worksheet Help system.
3. Choose **OK.**
4. Once you have connected to the OLAP Services instance, attach the analytic workspace that contains the program that you want to debug. To attach an analytic workspace, enter a DATABASE ATTACH command in the command input window in OLAP Worksheet.

   Example:

   ```
   database attach demo
   ```

5. Add your program to the trace list by entering the TRACE command in the command input window.

Example:

```
trace quarter.rpt
```

For more information on the TRACE command, see "OLAP DML Debugger Commands" on page 9-6.

6. Run your program by entering the program name in the command input window.

   Example:

   ```
   quarter.rpt
   ```

   When your program runs, the debugger suspends the program's execution according to the settings you have specified and displays the double-line arrow prompt in the output pane of the command input window.

7. Either select the debugger commands from the Debug menu or enter the debugger commands in the command input window to examine your program's execution. For more information on using the debugger commands, see "OLAP DML Debugger Commands" on page 9-6.

8. Leave the debugging environment by issuing the GO debugger command without any argument to have the program complete execution. If necessary, type `go` again to leave the debugging environment. The prompt returns to its single-line form (`->`) in the output pane.

OLAP DML Debugger Commands

Accessing the debugging environment

Access to the debugging environment is through the TRACE command, which controls the list of programs that is traced. To use the TRACE command, you must be connected to an analytic workspace from OLAP Worksheet. You enter OLAP DML commands in the command input window in OLAP Worksheet.

To debug a single program, attach the analytic workspace in which the program resides, and use the TRACE command to add that program to the trace list.

```
trace quarter.rpt
```

To trace all programs that are executed in your host session, specify an asterisk (`*`) instead of a program name.

```
trace *
```

To see the current trace list, which contains the names of all the programs that will be traced by the debugger when it is executed, specify a question mark (?) instead of a program name. For each program, the list contains the current TRACE settings.

```
trace ?
```

The following output is displayed in the output pane in the command input window.

```
QUARTER.RPT IN STOP ARGS OUT STOP VALUE
TRACE * IN STOP ARGS OUT NOSTOP VALUE
```

Using the debugger

When you run a program on the trace list, the debugging environment takes over execution of the program. When this happens, a double-line arrow prompt (=>) is displayed in the output pane of the command input window in OLAP Worksheet.

The debugging environment suspends execution of the program so that you can do the following:

- Step through a program one line at a time
- Display current values of OLAP DML objects
- Display the program's return value

You can use almost any OLAP DML command in the command input window. For example, you might use SHOW to examine the current values of OLAP DML objects.

Using the four debugger commands

You can also use four special debugger commands to help you investigate the current execution environment. These commands are listed in the following table.

Command	Description
GO	Specifies the number of lines that are executed in the debugging environment before execution stops. Each line of code is displayed as it is executed, except for lines that contain only comments. If you do not specify the number of lines, then all the code is executed to the end of the program without displaying the lines.
WATCH	Sets a watch point. When this point is reached in your program, execution is suspended. You can also use WATCH to enable, disable, or clear watch points. Finally, you can use WATCH to display a list of the current watch points.

Command	Description
BACK	Displays a list of currently running programs and indicates which programs call other programs. You have the option of displaying a program's arguments next to its name in the list.
ARGS	Displays a list of arguments for all the currently running programs. You can also display arguments for a single program.

For complete information about the program debugger commands, see the entry for TRACE in OLAP DML Reference.

Stepping through your program using the debugger GO command

When any program on the list is executed, the debugger suspends the program's execution according to the settings you have specified and displays the double-line arrow prompt in the output pane of the command input window.

At this point, you can use the GO debugger command to step through the execution of the program:

- To execute one line at a time, specify 1 as an argument.

  ```
  go 1
  ```

- To execute a group of lines, you can specify a higher number.

  ```
  go 3
  ```

At any point where the debugger prompts for a command, you can use OLAP DML commands to examine the current value of OLAP DML objects, or otherwise look at the execution environment to confirm that the program is executing as expected.

Note: Although the debugger displays labels in your program, it does not include them in the count of lines. Also, the debugger does not count or display lines that contain only comments.

BACK command

In many cases, you will trace a single program; however, you can use the OLAP Worksheet to trace a group of nested programs as well. The debugging environment will keep track of all the nested levels, and you have all the debugging capabilities in each program. To help keep track of the levels, you can use the BACK command to display a list of currently running programs. The list identifies the programs that call other programs and displays each program's arguments next to the program name.

Example: Using the BACK command in a debugging session

The following example presents a debugging session in which three nested programs are traced. PROG1 calls PROG2, and PROG2 calls PROG3. Only PROG2 accepts an argument, and the debugging environment displays its value upon entering PROG2. While in PROG3, the user types the BACK command to display the names of the programs that are currently executing, along with any arguments.

Enter the following commands in the input pane of the command input window in OLAP Worksheet:

```
trace prog1
trace prog2
trace prog3
prog1
```

The following is displayed in the output pane of the command input window:

```
= Entering PROG1
```

Enter the following command in the input pane:

```
go
```

The following is displayed in the output pane:

```
this is prog1
= Entering PROG2 From PROG1
= Args: 234
```

Enter the following command in the input pane:

```
go
```

The following is displayed in the output pane:

```
this is prog2
= Entering PROG3 From PROG2
```

Enter the following command in the input pane:

```
go
```

The following is displayed in the output pane:

```
this is prog3
= Leaving PROG3
```

Enter the following command in the input pane:

```
back
```

The following is displayed in the output pane:

```
= Command Level     -> PROG1
= PROG1             -> PROG2            (Args: 234)
= PROG2             -> PROG3
```

Enter the following command in the input pane:

```
go
```

The following is displayed in the output pane:

```
= Leaving PROG2
```

Enter the following command in the input pane:

```
go
```

The following is displayed in the output pane:

```
= Leaving PROG1
```

To exit the debugger, enter the following command in the input pane:

```
go
```

Exiting the debugging environment

From the Debug menu in OLAP Worksheet, choose Interrupt to exit the debugging environment.

Working with watch points

Setting a watch point using the WATCH command

If you do not want to trace the execution of a program from its beginning, then you can select a point in the program where you want to start tracing. You use the WATCH command to select a watch point. You can enter a WATCH command in the command input window in OLAP Worksheet.

The WATCH command sets a watch point as a Boolean expression that is monitored by the debugging environment. When the Boolean expression becomes true, the debugging environment suspends execution of the program and displays the double-line arrow prompt in the output pane of the command input window in OLAP Worksheet, from which you can examine the subsequent execution of the program.

To set a watch point, you specify the Boolean expression to be monitored, and, optionally, the name of a program in which to watch the expression. The Boolean expression can be something integral to your program, or you can include a watch point in the program just for debugging.

Example: Watching a Boolean expression that is integral to your program

In the sample program called QUARTER.RPT, a command limits the QUARTER dimension. You can use the results of that LIMIT command as a watch point as outlined below.

1. Because you do not want to trace execution of QUARTER.RPT from the beginning, remove it from the trace list.

   Example:

   ```
   trace quarter.rpt off
   ```

2. Set your watch point.

   Example:

   ```
   watch for statlen(quarter) eq 1 in quarter.rpt
   watch list

    1: Watch For STATLEN(QUARTER) EQ 1 within QUARTER.RPT ENABLED
   ```

   Note: The WATCH LIST command displays a list of the current watch points in the output pane in the command input window. When you set a watch point, it is assigned a watch number and is automatically enabled as indicated by the word ENABLED at the right end of the output line in the above example.

3. Run QUARTER.RPT. It executes until the status of QUARTER is limited to one value. Then the debugging environment suspends program execution and displays the double-line arrow prompt. You can step through the rest of the program or execute other OLAP DML commands in the command input window.

Enabling and disabling a watch point

Once a watch point expression is TRUE, the debugging environment disables it.

```
watch list
 1:Watch For STATLEN(QUARTER) EQ 1 within QUARTER.RPT DISABLED
```

You can enable it again with another WATCH command that specifies the watch number.

```
watch enable 1
```

Using a special debugging watch point

Instead of using an integral part of the program as a watch point, you can set up a special watch point specifically for the purpose of debugging.

Suppose you declare a local variable called WPOINT at the start of QUARTER.RPT and then set WPOINT to 1 at the point in the program where you want to start examining its execution. You can also set WPOINT to other values in other parts of your program to establish several watch points for different purposes.

```
describe quarter.rpt
DEFINE QUARTER.RPT PROGRAM
PROGRAM
argument userchoice month
variable wpoint integer      "Define watch point
trap on finish
pushlevel 'quarter'
push decimals month quarter
limit month to userchoice
wpoint = 1                   "Set watch point
limit quarter to month
report down district across product: -
   heading 'Total Sales' -
   total(sales, district product quarter)
finish:
poplevel 'quarter'
END
```

If you now set a watch point as follows and run QUARTER.RPT, then the debugger suspends execution at the point where WPOINT is set to 1 and displays the double-line arrow prompt.

```
watch for wpoint eq 1 in quarter.rpt
quarter.rpt
```

10

Using Embedded SQL

Chapter summary

In this chapter, you will learn how to write programs that copy data between an analytic workspace and a relational database.

List of topics

This chapter includes the following topics:

- Using Relational Data
- Obtaining Access to the Relational Database
- Supported SQL Commands
- Checking for Errors
- Fetching Data into an Analytic Workspace
- Declaring a Cursor
- Opening a Cursor
- Fetching the Selected Data
- Closing a Cursor
- Using Dimensions as Output Host Variables
- Writing OLAP DML Data to a Relational Table
- Matching Oracle9i Data Types
- Using the Special Features of an OCI Connection
- Example: SQL Program

Using Relational Data

Overview: How you can use the OLAP DML to analyze relational data

Oracle OLAP Services allows you to either move data from the relational database into an analytic workspace, or load data (for example, from a flat file) directly into an analytic workspace. In most cases, the relational database will be used as the primary data store, because it offers advantages related to manageability, scalability, and open access. Analytic workspaces provide support for some analytic features such as forecasting, models, and custom analytic functions defined using the OLAP DML.

To use analytic workspaces, they must be populated with data. The primary method of populating analytic workspaces is to load data from tables in the relational databases. You might also want to update tables with data generated using the OLAP DML. For example, you might load historical sales data from tables onto an analytic workspace, forecast sales data for future time periods, and commit the results of the forecast to tables in a data warehouse.

The SQL command in the OLAP DML is used to interact with the relational database using SQL statements. Using the SQL command, you can select data from tables and load it into analytic workspaces, commit data in an analytic workspace to tables, and perform most other operations supported by SQL.

Differences between the relational database and the OLAP DML models

Relational tables

A relational database stores information in tables organized by rows and columns. Each table contains a column or a combination of columns whose values uniquely identify each row. These unique values are called primary keys and help ensure the integrity of the data. When the values in a column match the values of another table's primary key, that column is called a foreign key. Relationships between tables are established through primary and foreign keys. You can select columns of data from different tables and view them together as long as the tables are related in this way.

OLAP DML variables contain multidimensional data

In contrast, when you use an analytic workspace, you define variables to store data. These variables are multidimensional. Each cell in a variable represents a unique combination of dimension values. Dimensions act as keys to variables. The OLAP

DML operates on multidimensional variables and dimensions in analytic workspaces.

Obtaining Access to the Relational Database

OLAP Services is always connected to the relational database

OLAP Services is always connected to its parent Oracle relational database instance. There is no need to establish a connecting using the OLAP DML. OLAP Services does not allow connections to Oracle instances other than the parent instance.

Overview: Using SQL statements in OLAP DML

SQL consists of statements that retrieve, delete, insert, change, and manipulate data stored in relational tables. You can use this embedded SQL in your OLAP DML programs.

OLAP DML SQL command

You can issue SQL statements from an OLAP Services session using the SQL OLAP DML command. Ordinarily, the command is used in an OLAP DML program, but you can also execute some SQL commands it interactively. The argument for the SQL command is a SQL statement.

```
SQL sql_statement
```

You can use almost any SQL statement that is supported by Oracle. For example, you can fetch data from relational tables and store it in OLAP DML dimensions and variables.

Using quotation marks

Wherever you would normally use double quotes (") in a SQL statement, you must use a single quote (’) in OLAP Services because the OLAP DML interprets a double quote as the beginning of a comment.

Special SQL syntax in OLAP DML

The OLAP DML evaluates SQL statements, either in whole or in part, before sending them to the relational database. Be sure to use the syntax described in this manual, rather than the syntax described in your SQL documentation.

Supported SQL Commands

What SQL statements can OLAP DML evaluate?

The OLAP DML evaluates the following SQL statements, either in whole or in part, before sending them to the relational database.

EXECUTE	DECLARE	FETCH
OPEN	CLOSE	PREPARE

Commands and options for general SQL use

The following OLAP DML options are available.

Option	Description
SQL	A command that issues SQL commands.
SQLCODE	An integer option that holds the returned value from the most recently attempted SQL operation.
SQLERRM	A text option that contains an explanatory message after an error has occurred.
SQLMESSAGES	A Boolean option that controls whether SQL error messages are written to the current outfile.

SQLBLOCKMAX option

The SQLBLOCKMAX option is an integer option that controls the maximum number of records OLAP Services retrieves from an Oracle database at one time. This option provides a means of fine-tuning the performance of data fetches.

Unsupported SELECT statement

When using SQL interactively, you would typically execute a SELECT command to produce a table of data. However, most host programming languages, including OLAP DML, must receive the data one row at a time instead of an entire table at once. For this reason, you cannot issue a simple SQL SELECT statement interactively in OLAP DML. Instead, you must define a cursor within an OLAP DML program.

Defining Transactions

SQL supports transaction processing in an OCI connection to Oracle9*i*. This support allows you to save or abort a series of statements that you have defined as a logical unit of work. This method of processing prevents partial updates; either all statements or none within the unit of work are executed.

A COMMIT statement saves changes made in the current transaction, and a ROLLBACK statement discards them. In your programs, you will want to write the logic so that the program uses COMMIT when it runs successfully and ROLLBACK when it ends in an error.

The OLAP DML accepts the following commands for transaction processing:

```
SQL COMMIT [WORK]
SQL ROLLBACK [WORK]
```

Checking for Errors

How the OLAP DML handles SQL errors

Although the OLAP DML will signal some SQL errors, it does not automatically signal an error when the 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 by the Oracle relational 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 following values:

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 neither 0 nor 100	An error has occurred.

SQLERRM option

SQLERRM 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.

The SQLMESSAGES option controls whether SQL messages are sent to the current outfile, which is usually the screen.

Procedure: Sending messages to the current outfile

To send SQL messages to the current outfile, issue the following command.

```
sqlmessages = yes
```

Fetching Data into an Analytic Workspace

Pretesting SELECT statements

Before writing the code to fetch data into an analytic workspace, 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 or SQL 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.

Tip: Use ORDER BY clauses where necessary so that the data for multidimensional variables is fetched with the slowest-varying dimension values first. Use GROUPBY

clauses to perform simple aggregation of the data to the level at which it becomes useful for data analysis.

Definition: Cursor

You cannot issue a SELECT statement in the OLAP DML. Instead, you must define a cursor using embedded SQL in an OLAP DML program. In the context of a query, a cursor can be thought of as simply a row marker in a 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.

Using cursors

You must declare and open a cursor from within a single OLAP DML program. Then you can fetch the data and close the cursor either in the same program or a different program.

Summary of cursor support

Several special SQL statements are used to define and use cursors.

The following commands are associated with cursors. Each of them is discussed in detail in a separate topic.

OLAP DML Command	Description
SQL DECLARE CURSOR	Contains a SELECT statement to identify the data to be retrieved and associates this selection with the name of a cursor.
SQL OPEN	Opens the cursor so it can be used in a FETCH statement.
SQL FETCH	Retrieves the data associated with the cursor and stores it in one or more OLAP DML objects.
SQL CLOSE	Closes the cursor so that the program can no longer access results through it.

Declaring a Cursor

DECLARE CURSOR statement

A DECLARE CURSOR statement associates a cursor by name with the results of a data query. In OLAP DML, it has the following syntax.

```
SQL DECLARE cursor-name CURSOR FOR select-statement
```

Cursor name requirements

A cursor name can consist of 1 to 18 alphanumeric characters or the symbols @, _, $, or #. A name containing @, $, or # must be enclosed in single quotes. The first character cannot be a number.

Example: Declaring a cursor

In the following example, the cursor declaration selects rows from a table named Products that has columns for product identification codes (PROD_ID) and descriptive labels (PROD_NAME). A third column, SUGGESTED_PRICE, is used in a WHERE clause to limit the returned rows to only those in which the suggested price is greater than $20.00.

```
sql declare highprice cursor for -
   select Prod_ID, Prod_Name from products -
      where Suggested_Price > 20
```

Re-using a cursor name

If you try to declare a cursor with the same name as one that is already declared, but with a different SELECT statement, then an error is signaled. You must first free the cursor with a COMMIT or ROLLBACK statement when you wish to associate it with a different selection.

Definition: Input host variables

Instead of providing literal values in the WHERE clause of a SELECT statement, you can use the values of input host variables. An input host variable is supplied by the host program as a parameter to the SELECT statement. Note that the value of an input host variable is assigned when the cursor is *opened*, not when it is *declared*.

An input host variable can be any expression preceded by a colon. However, if you specify a multidimensional expression, such as a variable or dimension, then the first value in status is used.

Using input host variables

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 following are 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: Using input host variables

The following program fragment 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.

```
variable set_price short
set_price = 20
sql declare highprice cursor for -
   select Prod_ID, Prod_Name from products -
      where Suggested_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. Otherwise, the command might be ambiguous and produce unexpected results. Place the parentheses around the input host variable *preceding* AND or 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.

Example: Using conjunctions in a WHERE clause

The following program fragment uses the values of two arguments to limit the range of values selected for the time dimension.

```
arg start_time date
arg end_time date
      .
      .
      .
sql declare addtime cursor for -
   select Timestamp from shipments -
      where Timestamp between :(start_time) -
      and :end_time
```

Opening a Cursor

OPEN statement

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.
- Cannot contain ampersand substitution.

The following is the syntax of the SQL command with an OPEN statement as an argument.

```
SQL OPEN cursor-name
```

Cursor's active set

The SQL OPEN command:

- Evaluates the input host variables (if any) used in the definition of the specified cursor.
- Determines the cursor's active set (that is, the rows that satisfy the SELECT statement).
- Leaves the cursor positioned before the first row of the active set.

A cursor's active set 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 cursor's active set.

Fetching the Selected Data

FETCH statement

You use a FETCH statement to retrieve data from a relational database. The FETCH statement advances the cursor position to each subsequent row of the cursor's active set and delivers the selected fields into OLAP DML objects.

The cursor must already be declared and open before you can use the FETCH statement.

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...]
```

In the above syntax, targets represents output host variables, which can be one or more of the following:

```
[MATCH] dimension
[APPEND position] dimension
variable|qualified data reference|relation|composite
```

If the output host variable is a dimension, and that dimension is preceded by a APPEND keyword, then the position that follows APPEND is one of the following:

```
AFTER dimension-value
BEFORE dimension-value
FIRST
LAST
```

This chapter describes how to use the SQL FETCH command. For details about this command, refer to the SQL FETCH entry in the OLAP DML Reference. For more information about understanding the syntax, refer to "Using Dimensions as Output Host Variables" on page 10-14.

Definition: Output host variables

The fetched data is brought into one or more output host variables. An output host variable is an OLAP DML object that will be used to store the data retrieved from the relational database.

Using output host variables

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.

The following are examples of expressions that can be used as output host variables.

Type of Expression	Example
Variable (database or local)	`:sales`
Dimension	`:geog`
Qualified data reference	`:units(geog 'G4' prod 'P15' time 'T23')`

Fetching dimension values

Whenever you fetch data into a dimensioned OLAP DML 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.

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 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.

Note: Be sure to fetch the dimension values *before* the values of the variable. Otherwise, the fetch will not loop through the dimension values.

Fetching null values

Null values in a relational table are equivalent to NAs. In OLAP DML variables, null values do not pose a problem; they appear as NAs. However, you cannot have

a dimension value of NA. Therefore, any rows that have a null value are discarded in a column being fetched into a dimension.

Host indicator variables in the INTO clause of a FETCH statement are not supported. (Indicator variables are required when using embedded SQL in C and COBOL, which do not normally allow unknown or missing values in a variable.)

To promote good performance, use LOOP

Use the LOOP keyword to promote good performance. In most cases, you can improve performance by using the LOOP keyword instead of using a WHILE loop. The following is an example of a WHILE loop:

```
while sqlcode eq 0
        sql fetch highprice into :prod, :prod_label
```

Instead of using a WHILE loop, use the LOOP keyword, as shown below:

```
sql fetch highprice LOOP into :prod, :prod_label
```

For more information about the LOOP keyword and how to use it, refer to the SQL FETCH entry in the OLAP DML Reference.

Example: Fetching descriptive dimension labels

This example shows the SQL commands used to retrieve dimension labels for the PROD dimension. The FETCH statement stores descriptive labels in a variable named LABELS.P. The dimension values have already been fetched from the PROD_ID column of the Products table to the PROD dimension. They are fetched again only to align the values in the PROD_NAME column with the appropriate dimension values. Note that the output host variables, PROD and LABELS.P, must already be defined as objects in the analytic workspace.

```
variable set_price short
set_price = 20
      .
      .
      .
sql declare highprice cursor for -
   select Prod_ID, Prod_Name from products -
      where Suggested_Price > :set_price
      .
      .
      .
sql fetch highprice loop into :prod, :labels.p
```

Closing a Cursor

CLOSE statement

After you have used a cursor to retrieve all the data in its active set, you should 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's declaration; it only renders the active set undefined.

The following is the syntax of the CLOSE statement.

```
SQL CLOSE cursor-name
```

COMMIT and ROLLBACK statements

A COMMIT or ROLLBACK statement closes all cursors and cancels all cursor declarations.

Using Dimensions as Output Host Variables

Syntax of FETCH statement

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.

```
SQL FETCH cursor LOOP INTO [MATCH|APPEND] dimension
```

MATCH 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.

In the following example, the MATCH keyword is omitted because it is the default value. The data in the PROD_ID column of the Products table corresponds to the PROD dimension values in the analytic workspace. An error is signalled when a value from the relational table does not match any value in the PROD dimension.

```
sql fetch highprice loop into :prod, :labels.p
```

APPEND keyword

With the APPEND keyword, all values that do not match are added to the list of dimension values.

APPEND has the following arguments, which you use to specify the position where the new values will be inserted. LAST is the default value.

```
AFTER value
BEFORE value
FIRST
LAST
```

Example 1: Adding values to the PROD dimension

In the following example, the APPEND keyword allows new values to be added to the PROD dimension.

```
sql fetch highprice loop into :append prod, :labels.p
```

Example 2: Adding values to the PROD dimension

In the next example, new dimension values are added to the PROD dimension before the value P11.

```
sql fetch highprice loop into :append before 'P11' prod :labels.p
```

Writing OLAP DML Data to a Relational Table

PREPARE and EXECUTE statements

You write data stored in OLAP DML variables into a relational table by using those variables as input host variables in your SQL statements. When writing multiple records, you should use the PREPARE and EXECUTE statements so that the same INSERT statement does not have to be recompiled for each row of data being sent to the table.

The syntax of the PREPARE and EXECUTE statements is shown below.

```
SQL PREPARE statement-name FROM sql-statement
SQL EXECUTE statement-name
```

PREPARE and EXECUTE must appear in the same program and cannot include ampersand (&) substitution. When an input host variable is a dimension, you must use a FOR command to loop over its values; otherwise, only the first value in status is sent.

Example: Inserting OLAP DML data into a relational table

Suppose that you have been using the OLAP DML to plan the introduction of a new product line, and now you want to add these new products to your relational database. You could copy this information from your analytic workspace into the appropriate relational table.

OLAP DML objects

The following are the OLAP DML objects used to store the data.

```
DEFINE PROD DIMENSION TEXT
DEFINE LABELS.P VARIABLE TEXT <PROD>
DEFINE SUGGEST.P VARIABLE SHORT <PROD>
```

Inefficient FOR loop

The following program fragment shows how you would use a FOR loop so that all product values currently in status are copied to a table named Products.

```
for prod
   do
      sql insert into products -
         values(:prod, :labels.p, :suggest.p)
      if sqlcode ne 0
         then break
   doend
```

Efficient precompiled code

The previous example will run much more efficiently when the INSERT statement is compiled with the PREPARE statement. The next example 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.

```
sql prepare write_products from -
   insert into products -
      values(:prod, :labels.p, :suggest.p)
      .
      .
      .
```

```
for prod
   do
      sql execute write_products
      if sqlcode ne 0
         then break
   doend
```

Example: Conditionally updating a relational table

You can also use the values of a multidimensional OLAP DML 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 following program fragment updates only those rows in the Products table where the values in the PROD_ID column match the PROD dimension values currently in status.

```
for prod
   do
      sql update products -
      set Suggested_Price = :suggest.p -
         where Prod_ID = :prod
      if sqlcode ne 0
         then break
   doend
```

Matching Oracle9*i* Data Types

Table of equivalents

Oracle9*i* uses a mix of conventional and unique native data types. The equivalent data types for OLAP DML dimensions and variables are listed in the following table.

Oracle9*i*	OLAP DML Dimensions	OLAP DML Variables
CHAR	TEXT [WIDTH n], ID	TEXT [WIDTH n], ID, BOOLEAN
DATE	DAY, WEEK, MONTH, QUARTER, YEAR, TEXT	DATE, TEXT
FLOAT	N/A	DECIMAL
LONG	N/A	TEXT
LONG RAW	N/A	N/A
NUMBER	N/A	INTEGER, SHORTINTEGER, DECIMAL, SHORTDECIMAL
RAW	N/A	N/A
ROWID	N/A	N/A
VARCHAR2	TEXT	TEXT

Date data

When writing OLAP DML text to a DATE column, the text must be in the default date format. You can use slashes (/) or hyphens (-) as separators, as well as spaces. If the data is in a different format, then you can use the Oracle9*i* TO_DATE function in your SQL INSERT command. In the following example, TODAY is a text variable containing the current date in the format of `"March 16, 1995"`.

```
sql insert into shipments(Timestamp) -
     values(TO_DATE(:today, 'Month DD, YYYY')
```

Refer to your Oracle9*i* documentation for information about using the TO_DATE function.

Text data

You can retrieve the entire contents of VARCHAR2 and LONG columns into OLAP DML TEXT variables. The maximum size in Oracle9*i* is 2K bytes for VARCHAR2

columns and 2 gigabytes for LONG columns. Newline characters are inserted in an output host variable whenever it encounters carriage returns (ASCII code 13) or line feeds (ASCII code 10). A newline character is inserted whenever its maximum line length of 498 bytes is reached.

LONG data

To insert more than 2k bytes of text data into a LONG column, use the WIDE keyword before the name of the input host variable. With this keyword, the maximum size is 65k bytes. Longer values are truncated.

The following is the syntax of the WIDE keyword.

```
:WIDE input-host-variable
```

Data is passed with the WIDE keyword to Oracle9*i* as LONG data. (Otherwise, text data is passed as VARCHAR2.) Oracle9*i* imposes some restrictions on the LONG data type. An error will not be signaled when you violate these restrictions. However, you might get unexpected results.

Refer to the Oracle9*i* manuals for information about restrictions on the LONG data type.

Input host variables

The input host variable must have a TEXT data type. The following example shows the contents of an OLAP DML variable, TEXTVAR, being inserted into a LONG column, COL1 of TABLE1.

```
sql insert into table1(col1) values(:wide textvar)
```

Formula for calculating the number of characters

You can calculate the number of characters that will be sent to Oracle9*i* from an input host variable by using the following formula.

```
NUMCHARS(variable) + 2 * (NUMLINES(variable) - 1)
```

For example, the following command shows the number of characters that will be sent using BIGVAR as the input host variable.

```
show numchars(bigvar) + 2 * (numlines(bigvar) -1)
```

This formula counts the extra carriage return and line feed characters that have been inserted between lines when passing the text to Oracle9*i*.

Related information

For more information about OLAP DML data types, search OLAP DML Reference for the "DEFINE command." For descriptions of Oracle9*i* data types, see the *SQL Language Reference Manual* or the *PL/SQL User's Guide and Reference*.

Using the Special Features of an OCI Connection

Overview: Using the special features of an OCI connection

When you have a direct connection to an Oracle9*i* database, you can use some of its special features.

FOR UPDATE clause

Oracle9*i* has an additional FOR UPDATE clause in the SELECT statement. This syntax is supported in a cursor declaration so that you can update or delete data associated with the cursor.

WHERE CURRENT OF cursor clause

The WHERE CURRENT OF *cursor* clause is supported in UPDATE and DELETE statements for interactive modifications to a table.

Savepoints

Savepoints are ignored. If your program sets a savepoint and issues a ROLLBACK TO *savepoint* command, then all SQL commands are rolled back in the current transaction, and the transaction is ended.

Requirements for stored procedures and triggers

Support is provided for Oracle9*i* stored procedures and triggers. They cannot contain SELECT statements. An OLAP DML stored procedure cannot contain output variables or transactions, nor can it call another procedure.

OLAP DML syntax differs slightly from the standard Oracle9*i* syntax. A tilde (~) is required instead of a semicolon as a terminator, and two colons are required instead of one in an assignment statement.

You can create a stored procedure or trigger in an OLAP DML program.

Example: Creating a stored procedure

The following example shows the OLAP DML syntax for creating a procedure named NEW_PRODUCTS.

```
sql create procedure new_products -
   (id char, name char, cost number) is -
      price number~ -
   begin -
      price ::= cost * 2.5~ -
      insert into products -
         values(id, name, 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.

Examples: Running the sample procedure

Providing literal values for arguments

The following command uses the NEW_PRODUCTS procedure to insert a single row in the Products table.

```
sql procedure new_products -
   ('P81', '8mm Camcorder', 279.58)
```

Using OLAP DML data for arguments

The ADD_PRODS program runs the same procedure but inserts data stored in OLAP DML dimensions and variables into the Products table. A FOR loop is required to loop over all of the values in status.

```
DEFINE ADD_PRODS PROGRAM
LD Add new products using stored procedure NEW_PRODUCTS
PROGRAM
arg newprods text
```

```
trap on error
push prod
limit prod to &newprods
" Loop over PROD to insert the data
for prod
   do
      sql procedure new_products(:prod, :labels.p,-
         :cost.p)
      if sqlcode ne 0
         then break
   doend
if sqlcode ne 0
   then signal err1 'Insert data into table failed.'
" Save new data in Products table
sql commit
show 'New products added to products table'
goto cleanup
ERROR:
" Discard changes to Products table
sql rollback
show 'No new products added to Products table.'
CLEANUP:
pop prod
END
```

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: SQL Program

The sample relational database

The sample relational database used for SQL examples contains corporate data for a fictitious electronics company. All of the examples are derived from four tables: Stores, Products, Shipments, and Shipments_Detail.

The examples show selected data from these tables as displayed using SQL*Plus.

Stores table

The Stores table has a row for each retail outlet. The STORE_ID column is the primary key and uniquely identifies each store.

The following table shows sample data from three columns of the Stores table.

```
SQL> select store_id, store_name, city from stores;

STORE_ID STORE_NAME            CITY
-------- --------------------- ------------
G4       Sears                 Boston
G5       Cambridge Sound       Cambridge
G6       New England Stereo    Boston
G7       Tweeter               Burlington
G8       Tweeter               Nashua
            .
            .
            .
```

Products table

The Products table contains a row with information about each product. The PROD_ID column is the primary key and uniquely identifies each product.

```
SQL> select * from products;

PROD_ID    PROD_NAME             SUGGESTED_PRICE
---------- --------------------- ---------------
P9         CD Player                      248.95
P10        Receiver                       298.95
P11        Amplifier                      189.95
P12        Cassette Deck                  159.95
P15        Color TV                       499.95
P16        B & W TV                        74.95
             .
             .
             .
```

Shipments table

The Shipments table contains a row for each purchase order.

The ORDER_NO column is the primary key.

The STORE_ID column is a foreign key and contains store identification numbers listed in the STORE_ID column of the Stores table. The STORE_ID columns of the Shipments and Stores tables create a relationship between the two tables.

The same store identification numbers can appear in numerous rows, once for each order placed by a particular retail outlet. In the following sample of data from the Shipments table, store G16 appears twice.

```
SQL> select * from shipments;

ORDER_NO     STORE_ID CUST_PO         TIMESTAMP
------------ -------- -------------   ---------
OR25         G6       2FD963718       12-MAR-96
OR26         G20      2243-3122-05    12-MAR-96
OR27         G21      379711          12-MAR-96
OR28         G11      2332628347      12-MAR-96
OR29         G13      786219190       19-MAR-96
OR30         G16      163222          19-MAR-96
             .
             .
             .
```

Shipments_Detail table

The Shipments_Detail table identifies the quantity and type of items ordered.

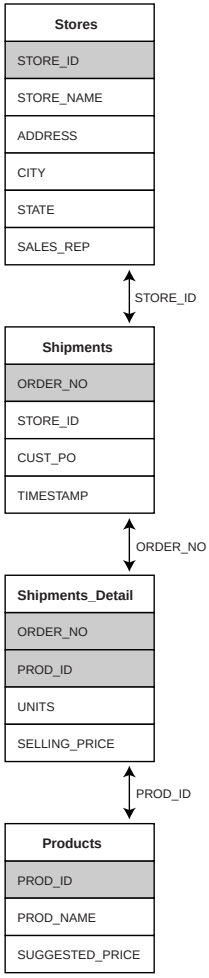
The primary key is composed of two columns, ORDER_NO and PROD_ID. The ORDER_NO column is also a foreign key containing values from the ORDER_NO column of the Shipments table. The ORDER_NO columns of the Shipments and Shipments_Detail tables create a relationship between these two tables.

The PROD_ID column is also a foreign key containing values from the PROD_ID column of the Products table. The PROD_ID columns of the Products and Shipments_Detail tables create a relationship between these two tables.

```
SQL> select * from shipments_detail where units < 100;

ORDER_NO     PROD_ID    UNITS      SELLING_PRICE
------------ ---------- ---------- -------------
OR51         P10        54         14528.97
OR51         P11        73         12479.71
OR51         P12        76         10940.58
OR51         P19        59          9555.35
OR51         P23        83         59756.27
OR51         P24        55         32172.52
OR52         P9         76         17028.18
OR52         P11        74         12650.67
             .
             .
             .
```

Diagram of relational database

Following is a diagram of the database. The primary keys are shaded and the relationships between tables are identified by arrows between them.



The sample analytic workspace

The SQL examples in this chapter use the tables shown in the previous topic to create an OLAP DML data variable named V.UNITS. It is dimensioned by

geography, product, and time dimensions, which are named GEOG, PROD, and TIME, respectively.

Since the relational tables provide codes for both stores and products, the Geography and Product dimensions use these codes as their dimension values. The full names for the stores and products are stored in separate text variables so that they are available for labeling the data in reports, but they are not used for storing and selecting data.

In the relational database, the order number is critically important for tracking individual sales. However, it is not an important factor from the standpoint of analytical processing. This information is not retrieved into the analytic workspace.

Geography dimension

The values for GEOG, the geography dimension, are fetched from the STORE_ID column of the Stores table. The GEOG dimension contains identifications numbers for all the stores, since the numbers are obtained from a primary key. The definition for the GEOG dimension is shown below, followed by a report of the first five values in GEOG.

```
DEFINE GEOG DIMENSION TEXT
LD All stores

GEOG
--------------
G11
G12
G13
G14
G15
```

Product dimension

The product dimension, PROD, also acquires its values from a primary key to get a complete list of all products. The product values are fetched from the PROD_ID column of the Products table. The definition for the PROD dimension is shown below, followed by a report of the first eight values in PROD.

```
DEFINE PROD DIMENSION TEXT
LD All products

PROD
--------------
P10
P11
```

```
P12
P15
P16
P17
P19
P20
```

TIME dimension

Time dimension values are fetched from the TIMESTAMP column of the Shipments table. The definition for the TIME dimension is shown below, followed by a report of the first eight values in TIME.

```
DEFINE TIME DIMENSION WEEK ENDING SATURDAY
LD Time dimension

TIME
--------------
W6.95
W7.95
W8.95
W9.95
W10.95
W11.95
W12.95
W13.95
```

Notice that the date values from the relational table were converted to a weekly format when they were brought into the TIME dimension in the analytic workspace.

V.UNITS variable

The number of items sold is fetched from the UNITS column of the Shipments_Detail column and stored in a variable named V.UNITS. This variable is dimensioned by GEOG, PROD, and TIME. Here is the object definition for V.UNITS.

```
DEFINE V.UNITS VARIABLE SHORTINTEGER <GEOG PROD TIME>
LD Units sold
```

The report command

```
report down prod across time: v.units
```

produces the following report.

```
GEOG: G7
                ------------V.UNITS-------------
                --------------TIME--------------
PROD              W3.96      W8.96      W11.96
--------------- ---------- ---------- ----------
P10                   185        137        127
P11                   153        153        155
GEOG: G11
                ------------V.UNITS-------------
                --------------TIME--------------
PROD              W3.96      W8.96      W11.96
--------------- ---------- ---------- ----------
P10                   455         NA        180
P11                   854         NA        773
```

Rotating the data cube

You can easily rotate a multidimensional data cube for different types of analyses. In the next example, products identify the rows, and stores identify the columns. Each time period appears in a separate report. The data is much easier to identify this time because the report shows the full descriptive labels for stores and products.

The report command

```
report w 20 down labels.p across labels.g: v.units
```

produces the following report.

```
TIME: W3.96
                     ------------V.UNITS-------------
                     --------------GEOG--------------
                     Sams Club  Tweeter -  Wal-Mart -
LABELS.P             - New York Burlington  New York
-------------------- ---------- ---------- ----------
Receiver                    455        185        166
Amplifier                   854        153         63
Cassette Deck               347         63        175
Color TV                    530        127        129
B & W TV                    436        189        134
Portable TV                 217         64        174
Standard VCR                670        170        110
Stereo VCR                  813        161        151
```

Summary of sample OLAP DML objects

The following table provides a summary of the OLAP DML objects that are created from the sample relational database.

OLAP DML Object	Created from. . .
Geography dimension (GEOG)	STORE_ID column of Stores table
Product dimension (PROD)	PROD_ID column of Products table
Time dimension (TIME)	TIMESTAMP column of Shipments table
Units variable, dimensioned by geography, product, and time (V.UNITS)	UNITS column of Shipments_Detail table
Geography labels variable, dimensioned by geography (LABELS.G)	STORE_NAME column of Stores table
Product labels variable, dimensioned by product (LABELS.P)	PROD_NAME column of Products table

Fetching PRODUCT dimension values and labels

The GET_PRODUCTS program fetches product codes into the PROD dimension and fetches descriptive labels into a text variable named LABELS.P.

Both PROD and LABELS.P have been defined in the database, but they do not yet have values.

The SELECT statement of the SQL DECLARE CURSOR command compares the value of a local variable with the suggested price for the product to determine whether or not to include a product in the fetch. Notice that you do not need to use the DISTINCT keyword in the SELECT statement; duplicate values from the fetch are disregarded when adding dimension values.

Object definitions

The object definitions are listed below.

```
DEFINE PROD DIMENSION TEXT
LD Product dimension
DEFINE LABELS.P VARIABLE TEXT <PROD>
LD Product labels
DEFINE GET_PRODUCTS PROGRAM
LD Get product dimension values and labels
PROGRAM
variable set_price short
```

```
set_price = 2
trap on error
" Connect to database if communications
" have not been established already.
.
      .
      .
" Declare a cursor named HIGHPRICE
sql declare highprice cursor for -
   select Prod_ID, Prod_Name from products -
      where Suggested_Price > :set_price
if sqlcode ne 0
   then signal err1 'Declare cursor failed.'
" Open the cursor
sql open highprice
if sqlcode ne 0
   then signal err2 'Cursor open failed.'
" Fetch the data into the PROD dimension and
" LABELS.P variable
sql fetch highprice loop into :append prod, :labels.p
if sqlcode ne 0 and sqlcode ne 100
   then signal err3 'Fetch failed.'
" Close the cursor
sql close highprice
if sqlcode ne 0
   then signal err4 'Cursor close failed.'
show 'Product dimension created.'
update
goto cleanup
ERROR:
show 'Product dimension build failed.'
CLEANUP:
" Free the cursor
sql rollback
END
```

Fetching GEOGRAPHY dimension values and labels

In this example, two columns of information are needed to describe the geography values uniquely. Since many of the stores are located in more than one city, the store name and the city must be combined into a unique descriptive label.

The GET_GEOLABELS program fetches the store and city names into temporary text variables first. Then the JOINCHARS function combines them into text for the

LABELS.G variable. The temporary variables are discarded at the end of the program, regardless of whether or not the program completes successfully.

Object definitions

The object definitions are listed below.

```
DEFINE GEOG DIMENSION TEXT
LD Geography dimension
DEFINE LABELS.G VARIABLE TEXT <GEOG>
LD Store and city labels
```

GET_GEOLABELS program

The GET_GEOLABELS program is listed below.

```
DEFINE GET_GEOLABELS PROGRAM
LD Get store and city names for geography labels
PROGRAM
define store.g variable text <geog> temp
define city.g variable text <geog> temp
trap on ERROR
" Connect to database if communications
" have not been established already.
      .
      .
      .
" Declare a cursor named GEOLABELS
sql declare geolabels cursor for -
   select Store_ID, Store_Name, City from stores
if sqlcode ne 0
   then signal err1 'Declare cursor failed.'
" Open the cursor
sql open geolabels
if sqlcode ne 0
   then signal err2 'Open cursor failed.'
" Fetch the data into the GEOG dimension and
" temporary variables STORE.G and CITY.G
sql fetch geolabels loop into :append geog, :store.g, :city.g
if sqlcode ne 0 and sqlcode ne 100
   then signal err3 'Fetch failed.'
" Close the cursor
sql close geolabels
if sqlcode ne 0
   then signal err4 'Close cursor failed.'
" Maintain the permanent LABELS.G variable
```

```
labels.g = joinchars(store.g, ' - ', city.g)
delete store.g city.g
update
show 'Geography dimension and labels maintained.'
goto cleanup
ERROR:
delete store.g city.g
show 'Geography dimension build failed.'
CLEANUP:
" Free the cursor
sql rollback
END
```

Maintaining the TIME dimension

The GET_TIME program adds new values to the TIME dimension, which is already defined with dimension values. Two arguments define the range into which the new values must fall.

GET_TIME fetches the new values from the relational table into a dimension named NEWTIME instead of TIME. Consequently, if the fetch fails while values are being read into the analytic workspace, then the analytic workspace can be restored easily to its previous state. The new values are added to the TIME dimension, using a MAINTAIN MERGE command, only after all values have been fetched successfully.

Passing arguments to GET_TIME

To run the program, you issue a command like the following one.

```
call get_time ('01APR95' '30JUN95')
```

Date formats

The arguments in this program are defined with a DATE data type. These arguments are passed directly to the data source, so they must be in a date format that the data source understands. The format is controlled by the OLAP DML DATEFORMAT setting. Whenever a program uses a DATE argument, you must make sure that DATEFORMAT is set to an appropriate format.

You could, however, define the arguments with a TEXT or ID data type. When a program uses a TEXT or ID argument to pass a date, you must make sure to call the program with arguments in a date format that is acceptable to the data source.

In the sample GET_TIME program, the OLAP DML date format is the default `"01JAN95"` when the starting and ending dates are passed to the data source. Later

in the program, the date format changes to "January 1st, 1995". Most data sources do not accept this date format. A date value passed in this date format would cause the OPEN CURSOR statement to fail.

The GET_TIME program also creates descriptive labels for time periods by converting the dimension values into more descriptive text. This is done entirely in the OLAP DML, using the DATEFORMAT option and the CONVERT function.

The following table will give you an idea of the various ways date data can be formatted after being fetched into the analytic workspace.

Date Format	Produced by. . .
W12.95	Fetching a date value into an OLAP DML dimension with a WEEK data type.
25MAR95	Fetching a date value into an OLAP DML variable with a TEXT data type. The format is not affected by the current DATEFORMAT setting.
March 25th, 1995	Converting a date or text value (either W12.95 or 25MAR95) into a date with this format: '<mtextl> <dtl>, <yyyy>'

Object definitions

The object definitions are listed below.

```
DEFINE TIME DIMENSION WEEK ENDING SATURDAY
LD Time dimension
DEFINE NEWTIME DIMENSION WEEK ENDING SATURDAY
LD Intermediate Time dimension
DEFINE LABELS.T VARIABLE TEXT <TIME>
LD Time descriptions
```

GET_TIME program

The GET_TIME program is listed below.

```
DEFINE GET_TIME PROGRAM
LD Get time periods
PROGRAM
arg start_time date
arg end_time date
push dateformat
```

```
trap on error
" Connect to database if communications
" have not been established already.
.
      .
      .
" Declare a cursor named ADDTIME
sql declare addtime cursor for -
   select Timestamp from shipments -
   where Timestamp between :(start_time) and :end_time
if sqlcode ne 0
   then signal err1 'Declare cursor failed.'
" Open the cursor
sql open addtime
if sqlcode ne 0
   then signal err2 'Cursor open failed.'
" Fetch time periods into dimension newtime
sql fetch addtime loop into :append newtime
if sqlcode ne 0 and sqlcode ne 100
   then signal err3 'Fetch failed.'
" Close the cursor
sql close addtime
if sqlcode ne 0
   then signal err4 'Close cursor failed.'
" Add new time periods to permanent TIME dimension
maintain time merge newtime
" Create time labels in format of January 1st, 1995
limit time to newtime
dateformat = '<mtextl> <dtl>, <yyyy>'
labels.t = convert(time, date)
show 'Time dimension and labels maintained.'
update
goto cleanup
ERROR:
show 'Time dimension maintenance failed.'
CLEANUP:
" Free the cursor
sql rollback
pop dateformat
maintain newtime delete all
END
```

Fetching SALES data

The GET_UNITS program fetches data into a variable named V.UNITS, which is dimensioned by the GEOG, PROD, and TIME dimensions. V.UNITS is already defined as an OLAP DML object. Notice that the SELECT portion of the DECLARE CURSOR statement uses an ORDER BY clause listing the column with the slowest-varying dimension values first.

Sorting data for efficiency

This sort order fetches the data in basically the same order that it will be stored, as shown by the following table. The table shows the results of a SELECT statement that was issued using the interactive interface of a relational manager.

```
Store_ID Prod_ID   Timestamp                Units
-------- --------- ------------------------ ---------
G17      P10          Jan  7 1995  8:56AM       103
G19      P10          Jan  7 1995  9:02AM        54
G5       P10          Jan  7 1995  9:10AM        69
G17      P11          Jan  7 1995  9:14AM        74
G19      P11          Jan  7 1995  9:23AM        73
G5       P11          Jan  7 1995  9:49AM       197
            .
            .
            .
G51      P32          Oct 21 1995  4:57PM       205
G5       P33          Oct 21 1995  5:04PM       251
G51      P33          Oct 21 1995  5:09PM       257
G5       P9           Oct 21 1995  5:18PM        83
G51      P9           Oct 21 1995  5:22PM       105
```

Object definitions

The object definitions are listed below.

```
DEFINE V.UNITS VARIABLE SHORT <GEOG PROD TIME>
LD Units sold
DEFINE TIME DIMENSION WEEK ENDING SATURDAY
LD Time dimension
```

GET_UNITS program

The GET-UNITS program is listed below.

```
DEFINE GET_UNITS PROGRAM
LD Fetch units sold data
```

```
PROGRAM
trap on error
allstat
" Connect to database if communications
" have not been established already.
      .
      .
      .
" Declare a cursor named GRABDATA
sql declare grabdata cursor for -
   select Store_ID, Prod_ID, Timestamp, Units -
      from shipments, shipments_detail -
      where shipments.Order_No -
         = shipments_detail.Order_No -
      order by Timestamp, Prod_ID, Store_ID
if sqlcode ne 0
   then signal err1 'Declare cursor failed.'
" Open the cursor
sql open grabdata
if sqlcode ne 0
   then signal err2 'Open cursor failed.'
" Fetch new values into V.UNITS variable. Use
" dimensions to align data in V.UNITS.
sql fetch grabdata loop into :geog, :prod, :time,:v.units
if sqlcode ne 0 and sqlcode ne 100
   then signal err3 'Fetch failed.'
" Close the cursor
sql close grabdata
if sqlcode ne 0
   then signal err4 'Close cursor failed.'
update
show 'V.UNITS variable populated.'
goto cleanup
ERROR:
show 'V.UNITS data incomplete.'
CLEANUP:
" Free the cursor
sql rollback
END
```

Writing OLAP DML data to a relational table

The CREATE_PRODUCTS program defines a new table in the relational database and populates it with data from an analytic workspace. It uses PREPARE and

EXECUTE statements so that the INSERT statement will not have to be recompiled for each dimension value.

Object definitions

The object definitions are listed below.

```
DEFINE PROD DIMENSION TEXT
LD Products
DEFINE LABELS.P VARIABLE TEXT <PROD>
LD Descriptive product names
DEFINE SUGGEST.P VARIABLE SHORT <PROD>
LD Suggested price
```

CREATE_PRODUCTS program

The CREATE_PRODUCTS program is listed below.

```
DEFINE CREATE_PRODUCTS PROGRAM
LD Create Products table
PROGRAM
trap on error
" Connect to database if communications
" have not been established already.
      .
      .
      .
" Create the products table using appropriate
" data types for the relational manager
sql create table products -
   (Prod_ID          text data type, -
   Prod_Name         text data type, -
   Suggested_Price   decimal data type)
if sqlcode ne 0
   then signal err1 'Create table failed.'
" Prepare the INSERT statement with a statement name
" of WRITE_PRODUCTS
sql prepare write_products from insert into products -
   values(:prod, :labels.p, :suggest.p)
if sqlcode ne 0
   then signal err2 'Prepare table failed.'
" Begin transaction, if appropriate
      .
      .
      .
```

```
" Loop over PROD dimension to execute INSERT statement
" for each product value in status
for prod
   do
      sql execute write_products
      if sqlcode ne 0
      then break
   doend
if sqlcode ne 0
   then signal err3 'Insert data failed.'
" Save changes to the products table
sql commit
show 'Products table populated.'
return
ERROR:
"Discard all changes to the database
sql rollback
sql drop table products
show 'Products table discarded.'
END
```

Updating rows using a FOR loop

The UPDATE_PRODS program sets the status of the PROD dimension to a value specified when you run the program.

UPDATE_PRODS uses a FOR command to loop over the selected products. The SQL UPDATE command updates the SUGGEST.P column of the Products table with the values from an OLAP DML variable named SUGGEST.P.

Passing arguments to UPDATE_PRODS

To run the program, you would issue a command like the one shown here.

```
call update_prods ('last 5')
```

Object definitions

The object definitions are listed below.

```
DEFINE PROD DIMENSION TEXT
LD Products
DEFINE SUGGEST.P VARIABLE SHORT <PROD>
LD Suggested price
```

UPDATE_PRODS program

The UPDATE_PRODS program is listed below.

```
DEFINE UPDATE_PRODS PROGRAM
LD Update products table
PROGRAM
arg _prodlist text
trap on error
push prod
limit prod to &_prodlist
" Connect to database if communications
" have not been established already.
" Begin transaction, if appropriate
      .
      .
      .
" Loop over products in status to update table
for prod
   do
      sql update products set -
         Suggested_Price = :suggest.p -
         where Prod_ID = :prod
      if sqlcode ne 0
         then break
   doend
if sqlcode ne 0
   then signal err 'Update data in table failed.'
show 'Products table updated.'
goto cleanup
ERROR:
"Discard all changes to the products table
sql rollback
show 'Product table was not updated.'
CLEANUP:
pop prod
END
```

Precompiling for efficiency

To improve performance, you should modify this program to use PREPARE and EXECUTE, as shown in the following program fragment.

```
sql prepare new_prices from update products -
   set Suggested_Price = :suggest.p -
   where Prod_ID = :prod
      .
      .
      .
for prod
   do
      execute new_prices
      if sqlcode ne 0
         then break
   doend
```

11

Reading Data from Files

Chapter summary

This chapter describes how to read data from external files.

List of topics

This chapter 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

Definition: Data-reading commands

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. 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 OLAP DML 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 OLAP DML dimensions and variables according to the descriptions of the fields.
RECNO function	Reports the current record number of a file opened for reading.

Description: File I/O commands

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.

Definition: Data-reading programs

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.

Reading Files

Basic steps: Reading from files

While reading from a file, you can format the data from each field individually, and you can use Express functions to process the information before assigning it to an OLAP DML 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 Express objects.
4. Close the file.

Methods for reading a 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.
- The FILEVIEW command processes one record at a time.

Consequently, there are two basic methods for reading a file.

Method	Command	Description	Use
1	FILEREAD	Processes all records in a file automatically.	Use when all records in the file are the same.
2	FILENEXT and FILEVIEW	Reads and processes one record at a time from the file.	Use when there is more than one type of record in a 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	Method 1(FILEREAD)	Method 2(FILENEXT and FILEVIEW)
Initialization	`variable funit integer` `trap on error`	`variable funit integer` `trap on error`

Body	`funit = fileopen(-`
	`  'datafile' read)`
	`fileread funit`
	`  .`
	`  .`
	`  .`
	`fileclose funit`
		`funit = fileopen(-`
		`  '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 can not 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

Using the fileunit

The FILEOPEN function opens a file and returns an integer that uniquely identifies that file. This 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.

Using file identifiers

A file identifier is a character string that specifies a file stored on disk. The file identifier always includes the file name. In addition, other information might be required to specify the full path location of the file. The format of file identifiers is different on different operating systems.

In the Windows environment, the format in which you specify a file name depends on where the file is located:

- For files that are local to the computer on which OLAP Services is running, use DOS format.
- For files remote to the computer on which OLAP Services is running, use either DOS or UNC format, unless explicitly stated otherwise in the documentation. However, be consistent; all references to a given file must be in the same format.

Syntax: DOS file name format

DOS file name format is as follows.

```
[d:] [\][path\] filename[.ext]
```

The *d* argument designates a disk drive.

The *path* argument is a path of directory names separated by backslash (\) characters.

The *filename* argument is the name of the file.

The *ext* argument is a 1- to 3-character extension preceded by a period.

Syntax: UNC file name format

UNC file name format is as follows

```
\\host\share\[path\] filename[.ext]
```

The *host* argument designates the host system.

The share keyword designates a shared area on the host.

The *path* argument is a path of directory names separated by backslash (\) characters.

The *filename* argument is the name of the file.

The *ext* argument is a 1- to 3-character extension preceded by a period.

Specifying a file identifier as a text expression

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 OLAP DML object names or reserved words.

The backslash character (\) in file identifiers is of special importance. The backslash is interpreted as an escape character. Therefore, when specifying a file identifier as a text expression, you must use double backslashes to designate single backslashes.

In most cases, when parsing file names:

- Forward slashes are converted to backslashes. This feature allows you to specify one forward slash instead of two backslashes.
- Extra backslash characters are removed when they occur anywhere in a path name other than in the first two column positions.

Reading Data from Files

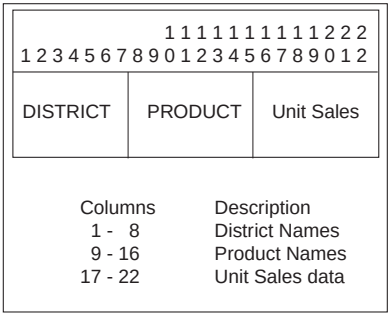
Using the FILEREAD command

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: Using a FILEREAD command in a data-reading program

File layout

Suppose you want to update unit sales data for the products in the `demo` analytic workspace. The new sales information is stored in a file called `units.dat`, which has the layout shown in the following figure.



FILEREAD command

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
```

How the FILEREAD command is processed

This command is processed as shown below.

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.

Program code

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('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: Reading a structured PRN file

File layout

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.

FILEREAD command

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

Reading records only for existing 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: Reading records only for existing dimension values

File layout

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
```

OLAP DML objects used by the program

The following OLAP DML objects are used by the example program.

```
DEFINE PRODUCTID DIMENSION ID
DEFINE PRODUCTNAME VARIABLE TEXT <PRODUCTID>
DEFINE UNITS.SOLD VARIABLE INTEGER <MONTH PRODUCTID>
```

Program description

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.

Program code

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('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: Adding new dimension values from a data file

Program description

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.

Program code

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('dr.dat' read)
fileread funit column 1 append width 6 productid
fileclose funit
funit = fileopen('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 or ID or a time data type (DAY, WEEK, MONTH, QUARTER, or YEAR) 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.

Example: Reading dimension values by position

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.

Converting time dimension values

When the records in a file contain values that represent time periods, you may need to specify a conversion type in the field description. The input values can be in any of the following formats:

- The VNF of the target time dimension, or the default VNF for the target dimension when it does not have a VNF of its own. For a VNF format, the conversion type is VNF, which is the default for time dimensions.
- Any of the input styles that are valid for dates. For date-style values, you must specify DATE as the conversion type.
- An integer that represents the number of days since December 31, 1899, or a negative integer for earlier dates (1 = January 1, 1900). For values in an integer-style date format, you must specify RAW DATE as the conversion type.

The FILEREAD command converts position values to time dimension values when you specify INTEGER as the conversion type.

Adding values to a time dimension

For input values that are in any format except position values, you can use the APPEND attribute to add values to a time dimension. Any gaps are filled in automatically between the values that already exist in your analytic workspace and the values that are read from the file. For example, when the MONTH dimension in your analytic workspace contains values for January 1995 through December 1996, and the file contains a value for March 1997, the months January 1997 through March 1997 are added automatically to the MONTH dimension.

Regardless of how time periods are represented in the file, they are used and stored internally as integers, and they are shown in output according to the VNF of the target time 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: 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. For time dimensions, the file values might not be in a format that can be interpreted as valid time periods for the target dimension. 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.

Translating dimension values with abbreviated codes

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.

Translating dimension values with complicated codes

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.

Using an assignment statement in the field description

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: Translating codes into dimension values

This example illustrates the use of an expression for translating codes into dimension values.

File layout

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
```

OLAP DML objects used by the program

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>
```

Program code

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('dr3.dat' read)
fileread funit -
   column 1 width 3 append rset '96' month
fileclose funit
funit = fileopen('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
```

Program description

The program translates the 2-character codes for districts and products into values of a DISTRICT dimension and a PRODUCT dimension. The program also illustrates how to translate 3-character month codes into values that can be recognized as valid time periods in a MONTH dimension.

When reading from the file, the program changes the 3-character month read from the file into a value that can be interpreted as a value of the MONTH dimension. Because a MONTH dimension value is identified by a month and a year, the program converts the format used in the file by attaching the year (96) at the end of each 3-character month value.

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.

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
the VNF command,	the topic for the command in the OLAP DML Reference
the DATEORDER option,	the topic for the option in the OLAP DML Reference

Processing Input Data

Modifying values read from a file

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 11-9.

Example: 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.

Example: Reading different fields for different types of records

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 an OLAP DML 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 as described earlier in this chapter.

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.

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
the FILEREAD command,	the topic for the command in the OLAP DML Reference
assignment statement,	"Assigning Values to Data Objects" on page 5-13. the topic for the = command in the OLAP DML Reference

Processing Records Individually

Reading records with varying types of data

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: Reading different data from the same record

File layout

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
```

OLAP DML objects used by the program

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>
```

Program description

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.

Program code

The example program, named DR.PROG4, contains the following code.

```
variable funit integer
variable varname text
funit = fileopen('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.

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
FILEPUT, FILENEXT, and FILEVIEW,	the topic for the command in the OLAP DML Reference

Processing Several Values for One Variable

Assigning multiple fields to the same 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.

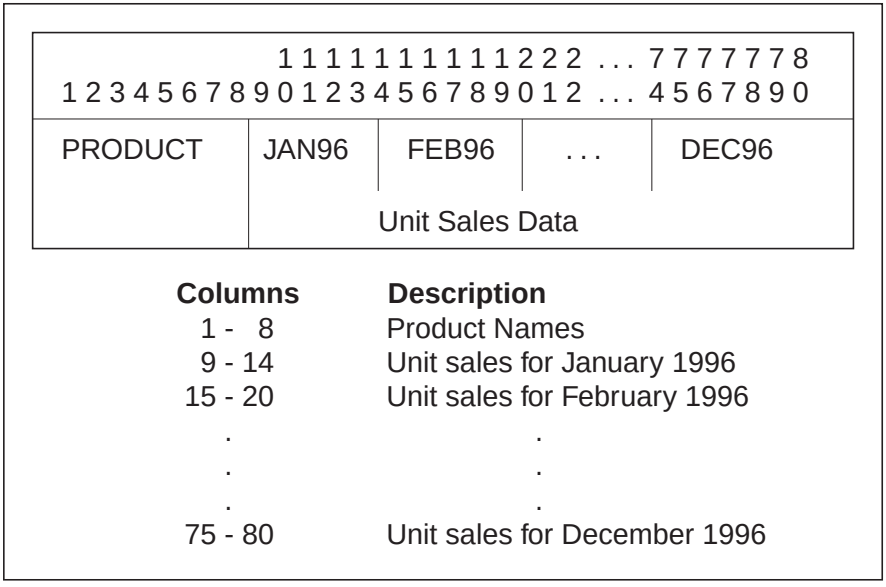
For repeating fields, you can use an ACROSS phrase in the field description to read the successive fields and place the values in the appropriate cells of the target variable. The ACROSS phrase extracts data for each dimension value in the current status or until it reaches the end of the record. You 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: Assigning multiple fields to the same variable

File layout

Successive fields might hold sales data for successive months, as shown in the layout of `unitsale.dat` in the following figure.



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.

Program code

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('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
```

Program description

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: Using input data to limit the ACROSS dimension

File layout

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
```

OLAP DML objects used by the program

The following OLAP DML objects are used by the example program.

```
DEFINE ENUM DIMENSION INTEGER
DEFINE MONTHNAME VARIABLE ID <ENUM> TEMPORARY
DEFINE SALESDATA VARIABLE DECIMAL <MONTH PRODUCT>
```

Program code

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('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
```

Program description

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.

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
the CHARLIST function,	the topic for the function in the OLAP DML Reference

12

Writing Reports

Chapter summary

This chapter explains how to create reports using the OLAP DML.

List of topics

This chapter includes the following topics:

- Introducing the Reporting Commands
- Creating Report Rows
- Creating Report Columns
- Retrieving Data for Rows
- Controlling the Default Format of Report Output
- Modifying the Layout of Columns
- Creating Headings
- Performing Calculations in a Report
- Creating Paginated Reports
- Creating Headings on Each Page
- Guidelines for Writing a Report Program

Introducing the Reporting Commands

Creating simple reports

Sometime you only need to quickly report the output for one or more data expressions. To produce a simple report you can use the REPORT command. This command automatically loops over the dimensions of the data and formats the output in a default layout. Output from the REPORT command is sent to the current outfile.

You can also customize the layout REPORT produces by using a number of options. In fact, because REPORT has an underlying format similar to ROW, all the options available with the ROW command (discussed later) are available with REPORT.

Sample simple report

You can create a simple report of the values of the STATE.CITY relation with the following command:

```
REPORT STATE.CITY
```

This command creates the following simple report

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

Creating complex reports

The OLAP DML includes a number of commands, functions, and options that you can use to write complex reports. The following table summarizes the commands, functions, and options that you can use to write reports.

Elements	Description
Commands	Produce rows of data and text. There are three key commands: ■ ROW — Produces rows of data ■ HEADING — Produces rows for titles, column headings, and side headings ■ BLANK — Produces blank rows Attributes in ROW and HEADING commands provide custom formatting for data and text output.
Format options	Control the default format of report output. The key format options include COLWIDTH, DECIMALS, LCOLWIDTH, and PARENS.
Calculation functions	Perform calculations on numeric data in a report. There are three key calculation functions: ■ COLVAL — Performs calculations on data in a row ■ SUBTOTAL — Calculates subtotals and totals for data columns ■ RUNTOTAL — Calculates running totals for data columns
Paging options	Provide paginated report output, and control features such as page size, margin size, and standard page headers. To produce paginated output, the PAGING option must be set to YES.

By combining these elements in a program, you can create custom reports with special formatting. For example, you can create a report with a different variable in each column or with a different variable in each row. You can format the data on each row exactly as you want, showing a different number of decimal places and appending different text (such as a dollar sign or percent sign) to each item in a row.

You can use special report calculation functions to perform calculations on the data in the rows and columns of a report.

To create a customized report, you combine the reporting commands, functions, and options with other OLAP DML commands in a program. Although you will use combinations of various commands in programs, you can experiment with most of the commands individually.

Creating Report Rows

Producing a simple row of data

The ROW command is used for producing each row of data in a report. In its simplest form, the syntax of the ROW command is as follows.

```
row expression
```

ROW followed by a data expression creates a row of output that contains the value of the expression.

Example: Showing a numeric value in a row

You can specify a numeric value as the *expression* argument to ROW. Suppose your application issues the following command.

```
row 100
```

The command produces the following output.

```
100
```

When you use the ROW command to produce a row of numeric data, each data value is right-justified in a column by default. “Creating Report Columns” on page 12-6 explains the default column format.

Example: Showing a literal text value in a row

If the *expression* argument to ROW is literal text, then you must enclose the text in single quotes. Suppose your application issues the following command.

```
row 'District'
```

The command produces the following output.

```
District
```

By default, text values are left-justified in a column.

Example: Showing a variable value in a row

You can specify a variable as the *expression* argument to ROW. Suppose your application issues the following command.

```
row units
```

The command produces a data value that corresponds to the first dimension values in status for UNITS.

```
200
```

Example: Showing a dimension value in a row

The *expression* argument to ROW can be a dimension. Suppose your application issues the following command.

```
row month
```

The command shows the current value of MONTH.

```
JAN95
```

Example: Showing a calculated value in a row

The *expression* argument to ROW can be a calculation. Suppose your application issues the following command.

```
row sales / units
```

The command produces output such as the following.

```
160.77
```

Example: Showing multiple values in a row

You can show more than one expression in a row by specifying each expression in a ROW command. Suppose your application issues the following command.

```
row district units sales
```

The command produces a row of output such as the following.

```
BOSTON                200  32,153.52
```

Creating blank lines

You can leave a blank line in a report by using the ROW command with no arguments. The following command produces one blank line.

```
row
```

Alternatively, you can use the BLANK command to leave a blank line. The following command also produces one blank line.

```
blank
```

To leave more than one blank line, you can provide a numeric argument to BLANK that indicates the number of blank lines you want to leave. The argument can be any integer expression with a value of zero or greater. If the argument is zero, then no blank lines are generated.

For example, to leave a series of three blank lines in a report, use the following command.

```
blank 3
```

Creating Report Columns

How columns are created

Even though each ROW command creates just one row of data, the data values are positioned in the row so that they will line up with the values in other rows of the report. Therefore, we refer to the space occupied by each data value as a column.

The ROW command places the first expression in the first report column, the second expression in the second column, and so on.

Creating columns for labels and data

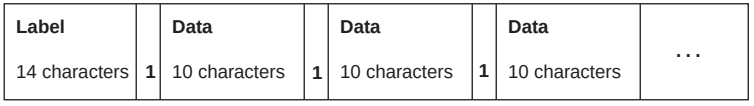
You can show any value in any column of a row, but the first column usually contains labels and the remaining columns usually contain data. Therefore, the first column is called the labels column. The remaining columns are called data columns.

Default layout of a report row

The default width of the labels column is controlled by the LCOLWIDTH option, which has a default value of 14 characters. The default width of data columns is controlled by the COLWIDTH option, which has a default value of 10 characters.

By default, text and date values in a column are left-justified, and numeric and Boolean values are right-justified. One space is left between columns.

A default row has the format shown in the next figure.



Leaving a column blank

You can use the SKIP keyword in place of an expression to leave a column blank. The command

```
row skip units sales
```

produces a blank column before the column of UNITS data.

```
200  32,153.52
```

Retrieving Data for Rows

No automatic looping in the ROW command

Unlike the REPORT command and many other OLAP DML commands, the ROW command does not automatically loop over the values of a dimension.

For example, if you show the DISTRICT dimension with a ROW command, then only the first district in status is shown. The command

```
row district
```

produces the following output.

```
BOSTON
```

Showing multiple values of a single expression in a row

You can use the ACROSS keyword with the ROW command to show multiple values of a single expression across a row. By specifying ACROSS followed by the name of one of the dimensions or composites of the expression, you can show a data value for each dimension value in the current status. The name of the dimension or composite must be followed by a colon.

Tip: You can specify an unnamed composite in an ACROSS phrase by using the syntax that was used to create it.

Example: Looping across the values of UNITS

The following commands create a row of output that shows the value of UNITS for each month in status.

```
limit month to 'JAN95' to 'MAR95'
row district, across month: units
```

These commands produce the following output.

```
BOSTON                  200         203         269
```

Example: Looping across the values of two expressions

You can apply the ACROSS keyword to more than one data expression by placing angle brackets around the expressions.

```
limit month to 'JAN95' 'FEB95'
row district, across month: <units sales>
```

The above commands create a row of output that shows a group containing a UNITS value and a SALES value for each month in status.

```
BOSTON                  200  32,153.52         203  32,536.30
```

Setting temporary dimension status within the row command

Instead of using a LIMIT command to set the status of a dimension before you execute a ROW command, you can temporarily set the status of the dimension within the ACROSS phrase of the ROW command. You specify a temporary status for the dimension by specifying any of the LIMIT keywords along with an appropriate value list or related-dimension list.

When you set the status for a dimension within an ACROSS phrase, the status remains in effect only for the duration of the ROW command.

For example, the following command temporarily limits the MONTH dimension to January 1995 and February 1995.

```
row district, across month to 'JAN95' 'FEB95': -
   <units, sales>
```

Handling a null status condition

You can use the IFNONE keyword in an ACROSS phrase to branch to a label if an attempt to set status would result in no values. Your report program might contain the following lines.

```
row district, across month keep units gt 500 -
   ifnone novals: units
   .
   .
   .
```

```
return
novals:
   .
   .
   .
```

Looping across composites

A variable that uses a composite includes either the SPARSE keyword or a named composite in its dimension list.

If you report data for a variable that uses a composite, and you do not include an ACROSS phrase in the ROW command, then ROW shows output for all data cells that correspond to the base dimensions of the composite. If a particular combination of base dimension values does not exist in the composite, then ROW shows NA for the corresponding data cell.

If you specify a composite in an ACROSS phrase, then ROW shows output only for data cells for which combinations of base dimension values exist in the composite. This gives you a more concise report that better reflects your data.

If you specify one of the composite's base dimensions in an ACROSS phrase, then ROW shows NA for a data cell for which the composite contains no value.

If you specify a composite in the ACROSS phrase of a ROW command, then you cannot specify LIMIT arguments. You must limit the base dimensions of a composite to the desired values before you execute a ROW command.

Creating a separate row for each dimension value

You can produce a separate row of output for each dimension value in the current status by executing a ROW command in a FOR loop for the dimension. The FOR command can only be executed in a program.

By using nested FOR loops in your program, you can loop over more than one dimension. For example, you might want to show sales data for each product within each district.

Example: Creating a separate row for each district

To show the data for each district in a separate row, you might include lines such as the following ones in your program.

```
limit month to 'JAN95' to 'MAR95'
limit product to 'TENTS'
```

```
for district
   row district, across month: units
```

The above commands loop over the DISTRICT dimension to create the following output.

```
BOSTON              200       203       269
ATLANTA             253       276       320
CHICAGO             181       181       247
DALLAS              297       313       419
DENVER              227       210       283
SEATTLE             271       257       322
```

Each row shows data for the TENTS product for each of the three months in the current status.

Example: Creating a separate row for each product within each district

You can loop over products within a loop over districts to create groupings of output rows like the ones shown below.

```
BOSTON
------
   TENTS              200       203       269
   CANOES             347       400       482
   RACQUETS           992     1,076     1,114
   SPORTSWEAR       1,096     1,214     1,294
   FOOTWEAR         2,532     2,405     2,775
ATLANTA
-------
   TENTS              253       276       320
   CANOES             260       285       356
   RACQUETS         1,037     1,196     1,158
   SPORTSWEAR       2,358     2,538     2,856
   FOOTWEAR         2,785     3,064     3,217
```

You can create the output shown above by using nested FOR loops in a program, as shown below.

```
limit month to 'JAN95' to 'MAR95'
limit district to 'BOSTON' 'ATLANTA'
limit product to all
for district
   do
   row under '-' valonly district
   for product
      row indent 3 product, across month: units
```

```
blank
doend
```

The first (or outer) FOR command in the above code loops over the values of DISTRICT and executes all the commands between the DO and DOEND commands. The second (or inner) FOR command loops over the values of PRODUCT to create the rows of data. Therefore, for each value of DISTRICT, rows of unit sales data are produced for each product. The inner FOR loop does not require the DO and DOEND commands because only one command is executed in the loop.

Tip: The ROW commands in this example use the format attributes UNDER, VALONLY, and INDENT. ROW command attributes are explained in the topic "Modifying the Layout of Columns" on page 12-12.

Controlling the Default Format of Report Output

Introducing format options

You can use several options that control the default format of report output. The following table lists the format options you will use most often.

Option	Description
COLWIDTH	Controls the width of data columns. Default: 10 characters.
DECIMALS	Controls the number of decimal places shown for DECIMAL values. Default: 2 places.
LCOLWIDTH	Controls the width of the labels column (the first column). Default: 14 characters.
PARENS	Controls whether negative numbers are represented with parentheses or a minus sign. Default: NO (a minus sign is used).

The effect of using format options

By changing the value of a format option, you change the format of output produced by subsequent ROW commands. If you want to use a specific format more or less consistently throughout your report, then you should set the relevant options to the appropriate values in the initialization section of your report program.

Example: Setting the DECIMALS option

If you want to suppress all decimals in your report, then include the following command in the initialization section of your report program.

```
decimals = 0
```

Overriding OLAP DML format options

After setting an option, you might find that you want to use a different format for an occasional row of data. In this case, you can specify a format attribute in the particular ROW command to override the value of the corresponding option.

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
on the format options,	search for the entries for the individual options in the OLAP DML Reference
table of all the OLAP DML format options that affect report output,	the entry for the ROW command in the OLAP DML Reference

Modifying the Layout of Columns

Key format attributes

You can specify format attributes within a ROW command to modify the column layout or the way the labels and data values are represented.

The following table lists the ROW attributes you will use most often.

Attribute	Abbreviation	Description
CENTER	C	Centers the value within its column.
DECIMAL *n*	D *n*	Shows *n* decimal places.
INDENT*n*		Indents the value *n* spaces within its column.
LEFT	L	Left-justifies the value within its column.
LSET'*text*'		Adds *text* to the left of a value.
OVER '*text*'		Overlines the column with the first character of *text*.
PAREN		Uses parentheses to indicate negative values.
RIGHT	R	Right-justifies the value within its column.
RSET '*text*'		Adds *text* to the right of the value.
SPACE *n*	SP *n*	Precedes the column with *n* spaces.
UNDER '*text*'		Underlines the column with the first character of *text*.
VALONLY		Underlines or overlines the value only, not the entire column; VALONLY is used only with UNDER or OVER.
WIDTH *n*	W *n*	Makes the column *n* characters wide.

Tip: For convenience, you can use abbreviations when they are available for an attribute. For example, you can abbreviate WIDTH as W.

The effect of using format attributes

The attributes in a ROW command affect only the row produced by that command. They have no effect on subsequent output rows created by other ROW commands, even when the same expressions are shown.

Overriding OLAP DML format options

The default values of some ROW attributes are controlled by OLAP DML format options. For example, the DECIMALS option determines the default value of the DECIMAL attribute in the ROW command.

If you specify an attribute in a ROW command, then the attribute overrides the value of the corresponding OLAP DML option.

Example: Modifying decimal places and column width

This example calculates the ratio of actual sales to budget sales. It shows a label in a label column that is 16 characters wide and a data value with four decimal places in a data column that is 6 characters wide.

```
row width 16 'Percent of Plan', decimal 4 -
   width 6 sales / sales.plan
```

The above commands produce a row of output such as the following.

```
Percent of Plan  0.7593
```

Example: Calculating the width of a data column

This example calculates the width of the UNITS column based on the current value of the LSIZE option. The command

```
row skip, w lsize / 5 units
```

produces the following output.

```
200
```

Example: Underlining values

The following command uses the hyphen character to underline the UNITS data column.

```
row skip, under '-' units
```

This command produces the following output.

```
                200
           ----------
```

Applying attributes to more than one expression

To apply an attribute to more than one expression in a row, you can specify the attribute once and enclose the expressions in angle brackets. Attributes that apply to more than one data expression are called global attributes.

Place the global attributes just before the angle brackets, then place any attributes that apply to just one expression within the angle brackets.

Tip: If you want to use the same format throughout most of your report, then you might be able to set a format option instead of specifying format attributes.

Example: Applying format attributes to two expressions

The following command sets the width of the columns for both SALES.PLAN and SALES to nine characters. In addition, it specifies a different number of decimal places for each data expression.

```
row district, w 9 <d 0 sales.plan, d 2 sales>
```

The above command produces the following output.

```
BOSTON            42,347 32,153.52
```

Using different attributes in different rows

You might want to use different format attributes for data in different rows of a report. For example, you might want to insert a dollar sign to the left of values in the first row of a report but not in subsequent rows.

You can store the desired attribute arguments in variables dimensioned by one or more dimensions of your data, and specify the variables as arguments to the format attributes in your ROW commands.

Example: Using different attributes in different rows

If your report shows sales data with districts going down the page, then you can insert a dollar sign for the sales value for the first district, BOSTON, but not for the other districts.

First, define a text variable named DOLLAR.SIGN, dimensioned by DISTRICT. In the DOLLAR.SIGN variable, store a dollar sign ($) for BOSTON and blank characters for the other districts. To enter a blank character in DOLLAR.SIGN, specify two single quotes with nothing between them, as shown below.

```
limit district to 'BOSTON'
dollar.sign = '$'
limit district complement
dollar.sign = ''
```

In your report program, specify DOLLAR.SIGN as the argument to the LSET attribute.

```
limit district to all
for district
   row across month: lset dollar.sign sales
```

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
ROW attributes, including a complete listing of attributes along with their default values and the corresponding OLAP DML options,	the entry for the ROW command in the OLAP DML Reference

Creating Headings

Overview of the HEADING command

You use the HEADING command to create titles, column headings, and side headings in reports. The HEADING command is like the ROW command except that any numbers shown in headings are not added in when you generate totals of the columns or rows in your report.

The HEADING command has the same syntax as the ROW command. You can use any of the ROW format attributes with the HEADING command to change the format of titles and headings.

The HEADING command creates rows of output with the same default layout as those produced by the ROW command. The first expression you specify in the HEADING command is shown in the labels column, and the remaining expressions are shown in the data columns.

Creating report titles

You use the HEADING command to create titles on your reports. To center a title on a page, you use the CENTER attribute along with a WIDTH attribute that specifies the column width as the width of the page.

Example: Centering a title

For example, to center a title on a page that is 54 characters wide, you can use the following command.

```
heading w 54 c 'Unit Sales'
```

This command produces the following title.

```
Unit Sales
```

Example: Incorporating text values in a title

Your title can be any character expression. For example, if you are creating a report for each district, then you can incorporate the current district in your title.

```
heading w 54 c joinchars('Unit Sales for ' district)
```

If the current district is Boston, then this command produces the following title.

```
Unit Sales for BOSTON
```

Creating column headings

You also use the HEADING command to create headings for columns of data. When you create column headings, you use the same layout that you use for the data columns. You can use any ROW command attributes to format the headings.

Tip: Be sure the format attributes in the HEADING command that creates the column headings match the format attributes in the ROW commands for the corresponding data columns in your report program. If the attributes do not match, then the headings will not line up properly with the data columns.

Example: Creating headings for month columns

If you report unit sales data for Boston for the first three months of 1996, then you will want to label the columns with the names of the months. You can use the following commands in a report program.

```
limit month to 'JAN96' to 'MAR96'
heading skip, across month: w 6 sp 4 c -
   under '=' valonly month
```

The following report headings are produced.

```
JAN96      FEB96      MAR96
=====      =====      =====
```

This example uses the WIDTH attribute to specify the width of the columns, CENTER to center the month values in the columns, UNDER to underline the month values, and VALONLY to underline just the month values and not all of the values.

Creating side headings

You can also use the HEADING command to create side headings in a report. For example, suppose your program loops over the DISTRICT dimension to create rows

of data for each district in status. You can use the HEADING command to generate a label for each district.

Example: Creating side headings for rows of data

Suppose that you have a program that loops over the DISTRICT dimension to create rows of data for each district. The following commands insert a row that contains the name of the district before the rows of data for each district.

```
for district
   do
   heading under '=' valonly district
   .
   .            "Row commands
   .
   doend
```

The commands produce output with the following format.

```
BOSTON
======
   .
   .
   .
ATLANTA
=======
   .
   .
   .
```

Formatting time dimension headings

The values of a time dimension are formatted according to the VNF attached to the dimension definition. Or, if a time dimension does not have a VNF of its own, then the values are formatted according to the default VNF for the type of time dimension you are using.

For additional flexibility in formatting the values of a time dimension in a report, you can override the dimension's VNF (or the default VNF) by using the CONVERT function with a VNF argument.

Example: Customizing column headings for MONTH

To override the VNF for the MONTH dimension when you create month headings in a report, you can use commands such as the following.

```
limit month to 'JAN96' to 'MAR96'
heading skip, across month: w 10 sp 2 c under '=' -
   convert(month text '<mtextl>')
```

The following headings are produced.

```
January     February    March
==========  ==========  ==========
```

Related information

For more information, see the following table.

IF you want documentation about . . .	THEN see . . .
the default VNFs and for instructions on how to assign a VNF to a time dimension,	the topic for the VNF command in the OLAP DML Reference
attributes that you can use with the HEADING command,	"Creating Headings" on page 12-16

Performing Calculations in a Report

Key report calculation functions

You can use three key report calculation functions to perform calculations on numeric data that you have already generated in the current report. These functions are summarized in the following table.

Function	Description
COLVAL	Enables you to make calculations that involve data values in the current row. Returns the value already generated in a specified column of the current row of a report. Returns a decimal value.
SUBTOTAL	Calculates subtotals and totals for columns of data generated in previous rows. Returns one of 32 subtotals that are maintained for the current column, and resets that subtotal to zero. Returns a decimal value.
RUNTOTAL	Calculates running totals for columns of data generated in previous rows. Returns the running total for the current column (like SUBTOTAL, except it does not reset the subtotal to zero). Returns a decimal value.

The report calculation functions make it easier and faster to create a report. By using the functions, you can often avoid recalculating expressions that have already been generated in preceding columns.

Tip: In addition to using the three report calculation functions to perform calculations on values in your report, you can use other OLAP DML functions, such as TOTAL and LAG, to access data in your analytic workspace.

Performing row calculations

You can use the COLVAL function to access a numeric data value in a previous column of the current report row and perform calculations on that value.

As the argument to the COLVAL function, you specify the column number you want to access. The labels column is column 1, and the data columns are numbered from left to right, starting with column 2.

You can also use a negative argument to refer to the position of a column relative to the current column.

Example: Calculating the ratio of two columns

To produce a row of output that shows actual dollar sales, budgeted dollar sales, and the ratio of the two, use the following commands.

```
allstat
row product, sales, sales.plan, colval(2)/colval(3)
```

This ROW command produces the following output.

```
TENTS          32,153.52  42,346.89       0.76
```

Example: Referring to the relative position of a column

The following command uses a negative argument to refer to the position of a column relative to the current column.

```
row product, sales, sales.plan, colval(-2)/colval(-1)
```

In the command above, COLVAL(–2) returns the value in the SALES column, and COLVAL(–1) returns the value in the SALES.PLAN column. The command produces the following output.

```
TENTS          32,153.52  42,346.89       0.76
```

Calculating column subtotals and totals

By using the SUBTOTAL function, you can create a row that shows the total of one or more numeric columns in a report.

Thirty-two subtotals are maintained for each column of numeric data in a report, which means you can include up to 32 levels of totals and subtotals in your report. As the argument to the SUBTOTAL function, you supply the number of the subtotal you want to access.

Each time you use the SUBTOTAL function to access one of the subtotals, the contents of that subtotal are reset to zero.

Example: Calculating product totals for each month

Suppose your program loops over the PRODUCT dimension to create this report of unit sales.

	JAN95	FEB95	MAR95
TENTS	200	203	269
CANOES	347	400	482
RACQUETS	992	1,076	1,114

By using the SUBTOTAL function, you can create a row that shows the total of each numeric column. The SUBTOTAL function returns decimal values, which are shown with two decimal places by default. However, because you are totaling integer data, you will want to format the totals with no decimal places. After the commands that produce the rows of data, execute the following command in your program.

```
row indent 2 'Total', across month: over '-' -
   under '=' d 0 subtotal(1)
```

The report looks like the following.

```
                    JAN95       FEB95       MAR95
                ========== ========== ==========
TENTS                200         203         269
CANOES               347         400         482
RACQUETS             992       1,076       1,114
                ---------- ---------- ----------
  Total            1,539       1,679       1,865
                ========== ========== ==========
```

Example: Calculating subtotals and totals

If your program loops over the DISTRICT dimension as well as over the PRODUCT dimension, then you can show subtotals for the products within each district and then a grand total for all the districts. You can use SUBTOTAL(1) for the total of the products within each district, and SUBTOTAL(2) for the grand total of all districts. The program might contain the following commands.

```
limit month to 'JAN95' to 'MAR95'
limit product to 'TENTS' 'CANOES' 'RACQUETS'
limit district to 'BOSTON' 'ATLANTA'
heading skip, under '=' across month: c month
blank
for district
   do
   row under '-' valonly district
   blank
   for product
      row product, across month: units
   row indent 2 'Total', over '-' across month: -
      d 0 subtotal(1)
   blank
   doend
row 'Grand Total', over '-' under '=' across month: -
   d 0 subtotal(2)
```

This program produces the following report.

```
                JAN95       FEB95       MAR95
              ========== ========== ==========
BOSTON
------
TENTS              200         203         269
CANOES             347         400         482
RACQUETS           992       1,076       1,114
              ---------- ---------- ----------
  Total          1,539       1,679       1,865
ATLANTA
-------
TENTS              253         276         320
CANOES             260         285         356
RACQUETS         1,037       1,196       1,158
              ---------- ---------- ----------
  Total          1,550       1,757       1,834
              ---------- ---------- ----------
Grand Total      3,089       3,436       3,699
              ========== ========== ==========
```

Calculating running totals for columns

You can use the RUNTOTAL function to calculate a running total of the values in a column. As the argument to RUNTOTAL, you supply the number of the subtotal you want to access. Unlike SUBTOTAL, the RUNTOTAL function does not reset the subtotal to zero when you access it. You must reset the subtotal yourself when you want to start a new running total. To reset the subtotal, you use the ZEROTOTAL command.

Example: Calculating cumulative sales

To show cumulative unit sales and dollar sales over several months, you might want to create a report such as the following one.

```
Month   Units      Total    Dollars      Total
                   Units               Dollars
------ ------ ---------- ---------- ----------
JAN96     307        307  50,808.96  50,808.96
FEB96     209        516  34,641.59  85,450.55
MAR96     277        793  45,742.21 131,192.76
APR96     372      1,165  61,436.19 192,628.95
MAY96     525      1,690  86,699.67 279,328.62
JUN96     576      2,266  95,120.83 374,449.45
```

To create the report, use these lines in a program.

```
limit month to 'JAN96' to 'JUN96'
limit product to all
limit district to 'BOSTON'
row under '-' r <l w 6 'Month', w 6 'Units', -
   'Total Units', 'Dollars', 'Total Dollars'>
for month
   row w 6 <month, units>, d 0 units + runtotal(1),-
     sales, sales + runtotal(1)
```

Initializing the subtotals

When you use a SUBTOTAL or RUNTOTAL function in your program, you should include a ZEROTOTAL command in the initialization section of the program. Including a ZEROTOTAL command will ensure that all the totals are set to zero before you begin your report. Before you start a new running total for a column, you must use the ZEROTOTAL command to reset the subtotal of the column to zero.

If you want to reset all the totals for all the columns to zero, then use the ZEROTOTAL command with no argument.

```
zerototal
```

To reset a particular subtotal in a particular column to zero, specify the subtotal and the column number as arguments to ZEROTOTAL. For example, to set the first subtotal of the second column to zero, use this command.

```
zerototal 1, 2
```

To reset all the totals in a specific column to zero, specify ALL as the subtotal argument to ZEROTOTAL. For example, if you want to reset all the totals for the second column to zero, then use this command.

```
zerototal all, 2
```

Showing data calculated by functions

In a report, you can show the result returned by any OLAP DML function. To show the result, you can use either of the following approaches:

- Specify the function in the expression to be reported.
- Create a variable that holds the result returned by the function, and then specify the holding variable in the expression to be reported.

Example: Storing a calculation in a holding variable

Suppose you want to show the ratio of the unit sales for each district to the total unit sales for all districts combined. You can use the following expression to make the desired calculation.

```
total(units, district) / total(units)
```

If you make the above calculation in a FOR loop for the districts in a report program, then the expression must be recalculated for each district. Your program will run more efficiently when you define a variable to hold the calculated total, and use a FOR loop to report the holding variable. First, define a holding variable.

```
define tot decimal <district> temp
ld Total units ratio by district
```

Then use the following lines in a program to calculate the ratio, expressed as a percentage.

```
limit month to 'JAN95'
limit product to all
limit district to all
   .
   .
   .
tot = total(units, district) / total(units) * 100
for district
   row district, d 1 tot
```

The above commands produce this output.

```
BOSTON              16.4
ATLANTA             21.2
CHICAGO             17.2
DALLAS              18.6
DENVER              15.6
SEATTLE             11.1
```

Creating Paginated Reports

Introducing the paging options

You can use a number of options to produce report output in separate pages and control the format of the pages.

The paging facility is controlled by the PAGING option. When you want to produce your report in separate pages, set the PAGING option to YES in the initialization section of your report program. When PAGING is set to YES:

- Output from commands such as ROW, HEADING, and REPORT is produced in separate pages.
- You can use other OLAP DML paging options to change page features such as the size of the page, the size of the margins, and the headings that are used on each page.

Summary of the paging options

The paging options are summarized in the following table.

Option	Description
BMARGIN	Specifies the number of blank lines for the bottom margin of output pages. Default: 1.
LINENUM	Holds the current line number for the current page of output. It is adjusted automatically as output lines are produced. You should generally set LINENUM to 1 when you want to produce a report starting on page 1.
LINESLEFT	Holds the number of lines left on the current page. It is adjusted automatically as output lines are produced.
LSIZE	Specifies the width of the line within which the STDHDR program centers the standard header. Default: 80.
PAGENUM	Holds the current page number of the output. It is adjusted automatically as lines of output are produced. You should generally set PAGENUM to 1 when you want to produce a report starting on page 1.
PAGEPRG	Holds the text of a command or the name of a program to be executed at the beginning of each page of output. The default, 'STDHDR', specifies the standard header program. For no headings, specify 'NONE'.
PAGESIZE	Specifies the number of lines in a page of output, including the top margin and bottom margin. Default: 66.
PAGING	Controls the paging facility. Default: NO.
TMARGIN	Specifies the number of blank lines for the top margin of output pages. Default: 2.

How the line number is calculated

When a new page is used to store data, LINENUM is set to 1 and PAGENUM is incremented. However, the first line of output is produced after the number of lines used by TMARGIN and the lines produced by the PAGEPRG program. By the second line of actual output, LINENUM is set to the actual line number on the page.

For example, if you are using the default page layout, and you show the value of LINENUM on the first line of the body of the page, then its value is 1. However, if you show LINENUM on the second line, then its value is 6 because the 2 lines for TMARGIN and the 2 lines produced by STDHDR have been added to the line count.

Using the PAGE command to start a new page

If the PAGING option is set to YES, then you can use the PAGE command to force a page break in a report. For example, you might want to start each section of data on a new page of the report.

Example: Showing data for each product on a separate page

For example, if your report shows regional sales data for each product, then you can start the data for each product on a separate page. Your report program might include the following lines.

```
for product
   do
   page
   row under '-' valonly product
   for region
      row indent 3 region, across month: units
   doend
```

The PAGE command forces a new page at the beginning of the FOR loop for each value of the PRODUCT dimension.

Example: Resetting the page number for each product

Suppose you want to restart the page numbering of your report at 1 for each new product. You can use the following code to reset the PAGENUM option to 0 before executing a PAGE command.

```
for product
   do
   pagenum = 0
```

```
page
row under '-' valonly product
for region
   row indent 3 region, across month: units
doend
```

Starting a new page based on the lines left

Your decision to start a new page might depend on the space available on the current page. For example, you might want to start a block of data on a new page when there is not enough room on the current page for the entire block.

To start a new page when fewer than a specific number of lines remain on the current page, test the value in the LINESLEFT option.

For example, to start a new page when fewer than 12 lines are left on the current page, use these commands in your program.

```
if linesleft lt 12
   then page
```

Sending output to different outfiles

You can use the OUTFILE command to specify a file as the target outfile or to direct output to the default outfile.

The following command sends output to a file named `repfile.txt`.

```
outfile 'repfile.txt'
```

The following command redirects output to the default outfile.

```
outfile eof
```

Unless you use the OUTFILE command to send output to a file, your default outfile is used.

Using paging options with different outfiles

The paging options can have different values for the default outfile and for a different file. When you use the OUTFILE command to direct output to a file, the values of the paging options that are currently in use for the default outfile are saved automatically, and reinstates them when you redirect output back to the default outfile.

When you direct output to a file, the paging options are reset automatically to their default values for output to the file. After executing the OUTFILE command, you

can set the paging options as desired for the file. However, the values of the paging options for the file remain in effect only as long as you continue sending output to the same file. If you use another OUTFILE command to direct output to a different outfile, then the current values for the file are not saved.

Therefore, if you want the paging options to have a particular value for a particular file, then you must set the options each time you use the OUTFILE command for that file. This is true even when you are appending output to an existing file.

Initializing the page number

The value of the PAGENUM option is reset to 1 when you execute an OUTFILE command to send output to a file. However, if you redirect output to the default outfile, then PAGENUM will contain the value it last held for the default outfile. To be sure your report starts with Page 1, you should set PAGENUM to 1 in the initialization section of your report program.

Pausing during report output

You can make your program pause while producing report output when the PAGING option is set to YES in your Express session.

Using the PAGEPAUSE option and the PAUSE command

The following table shows how to pause during the execution of a report program.

IF you want your program to pause . . .	THEN . . .
after producing each page of output,	set the PAGEPAUSE option to YES.
at other times during the execution of the program,	use the PAUSE command.

When a pause occurs as a result of the PAGEPAUSE option or the PAUSE command, program execution is suspended, and a message is displayed in the prompt area at the bottom of the screen. The default message is as follows.

```
Press <Enter> to continue.
```

Execution of commands continues when the user presses a key.

Customizing the message for a pause

You can replace the default pause message with a message of your own, as shown in the following table.

IF you want to customize the message . . .	THEN . . .
for the PAGEPAUSE option,	set the PAGEPROMPT option to the desired text.
for the PAUSE command,	supply the message as an argument to the PAUSE command.

Creating Headings on Each Page

Controlling the page heading with the PAGEPRG option

When the PAGING option is set to YES, a heading is inserted at the top of each page of output. The PAGEPRG option specifies the text of a command or the name of a program used for generating page headings. By default, the value of the PAGEPRG option is STDHDR, which means that the STDHDR program is used for generating headings.

Using the standard page heading

The STDHDR program, which is provided with the OLAP DML, generates a standard heading. The standard heading consists of the date and time on the left side of the page, and the page number on the right side of the page. One blank line is left after the heading to separate it from the rows of the report.

```
16Sept96  16:37:20                              Page 1

```

The STDHDR program uses the values of the TODAY and TOD functions to generate the date and time in the heading. These two functions return the current date and time from the time clock in the user's PC.

Adjusting the position of the page number

The STDHDR program uses the value of the LSIZE option to determine the width of the page, and then uses the width of the page for placing the page number at the

right side of the page. By changing the value of LSIZE, you can adjust the position of the page number in the standard header.

If your current outfile is a file, then LSIZE can have different values for the default outfile and the current outfile. Therefore, you should set the LSIZE option after executing your OUTFILE command.

Customizing the page heading

Instead of using the standard heading, you can specify an OLAP DML command or write your own program to produce page headings. Then set the PAGEPRG option to the text of the command or the name of the program.

Example: Customizing the date in the page heading

Suppose you want to include the date and page number on a report but not the time. You can write a program called DATEHDR to produce the headings you want.

```
DEFINE DATEHDR PROGRAM
LD Heading program to create date and page number
PROGRAM
push dateformat
dateformat = '<yy>/<mm>/<dd>'
heading w 8 today, sp 0 w lsize-12 r 'Page', sp 0 w 4 -
   pagenum
pop dateformat
END
```

To use the DATEHDR program for your report headings, set the PAGEPRG option in the initialization section of your report program.

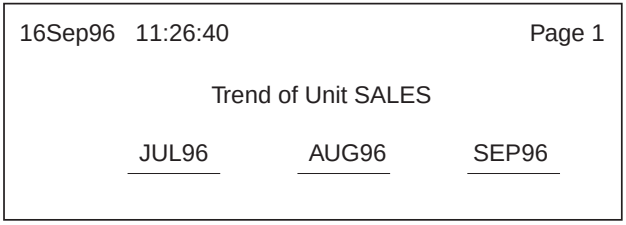
```
push pageprg
pageprg = 'datehdr'
   .
   .
   .
pop pageprg
```

Your report will have a heading at the top of each page that displays the date on the left and the page number on the right, as shown in the following illustration.

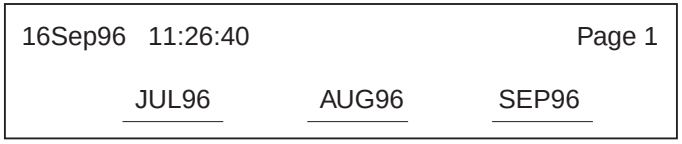
Example: Combining the standard heading with a custom page heading

Instead of replacing the standard heading with your own heading, you might want to use the standard heading and include additional headings of your own as well. For example, you might want to produce the standard heading on each page of the report, a title on the first page, and column headings on each page.

You might want the heading on the first page of the report to look like the one below.



You might want the heading on each successive page of the report to look like the one below.



You can use a program like HEAD.PRG to create the headings.

```
DEFINE HEAD.PRG PROGRAM
LD Produces std header, title, column headings
PROGRAM
stdhdr
if pagenum eq 1
   then do
      heading w lsize c 'Trend of Unit Sales'
      blank 2
      doend
heading skip, across month: under '-' c month
blank
END
```

For each page, the HEAD.PRG program first executes the STDHDR program to produce the standard heading. Then it tests the value of the PAGENUM option,

which holds the page number, and generates the report title on the first page only. Finally, the HEAD.PRG program inserts the column headings on each page.

In your report program, set the PAGEPRG option to HEAD.PRG.

```
pageprg = 'head.prg'
```

Tip: Be sure the format attributes in the HEADING command that creates the column headings match the format attributes in the ROW commands for the corresponding data columns in your report program. If the attributes do not match, then the headings will not line up properly with the data columns.

Guidelines for Writing a Report Program

Suggested outline of a report program

As a general rule, the commands in a report program should be placed in the sequence shown in the following table.

Program Section	Commands
Initialization	PUSHLEVEL command PUSH commands TRAP ON command ZEROTOTAL command Commands to set format options OUTFILE command PAGING command Commands to set paging options
Body	LIMIT commands ROW and HEADING commands
Normal Exit	OUTFILE EOF command POPLEVEL command RETURN command
Abnormal Exit	Error label Error-handling commands OUTFILE EOF command POPLEVEL command

In addition to organizing your report program as suggested in the above table, you should follow the suggestions that this manual provides for organizing and testing any type of program.

Tips on aligning report columns

When you test a report program, the problem you are most likely to encounter is that the report columns do not line up properly. You can avoid most column alignment problems by following these general rules:

- Use identical ACROSS groups in your HEADING and ROW commands.
- Use identical values of WIDTH and SPACE in your HEADING and ROW commands.
- When appropriate, you might also need to use identical values for INDENT, LEFT, RIGHT, and CENTER in your HEADING and ROW commands.

A

Creating and Using Analytic Workspace Metadata

Appendix summary

This appendix describes how to create and use analytic workspace metadata, which is necessary if you want to make the data that is contained in an analytic workspace viewable by the OLAP API.

List of topics

This appendix includes the following topics:

- What is Analytic Workspace Metadata?
- Analytic Workspace Metadata Prerequisites
- Metadata That Describes Dimension Hierarchies
- Metadata That Describes Dimension Hierarchy Levels
- Metadata That Describes Dimension Attributes
- Metadata That Describes Other Objects

What is Analytic Workspace Metadata?

Definition: Analytic workspace metadata

Analytic workspace metadata describes data in an analytic workspace to the OLAP API. If you wish to expose data, such as dimensions and measures, in an analytic workspace to the OLAP API, then the analytic workspace metadata must describe that data. You create analytic workspace metadata in an analytic workspace by

using the OLAP DML to create lower-level workspace objects and to set properties on these workspace objects.

Why is it necessary to create analytic workspace metadata?

When a Java client opens an analytic workspace, the data in that analytic workspace will not be visible unless you have created analytic workspace metadata.

If you follow the instructions in this appendix, then OLAP Services will make the analytic workspace metadata that you create available so that the OLAP API can display the analytic workspace data.

Where should the analytic workspace metadata be created?

The analytic workspace metadata should be created and stored in the analytic workspace in which data will be viewed by the OLAPI API.

What analytic workspace metadata needs to be created?

This appendix describes the metadata that you need to create. Typically, you need to set a property on an OLAP DML object that already exists in the analytic workspace. In addition, many properties will specify a metadata object that you must define and, in some cases, populate.

Analytic Workspace Metadata Prerequisites

Required analytic workspace metadata

To create analytic workspace metadata and make the OLAP API aware that the metadata exists, you must perform both of the following steps:

1. Define a metadata locator object.
2. Define metadata for every OLAP DML object that you wish for the OLAP API to display.

Defining the metadata locator object: ECMLOCATOR

The metadata locator object is the first point of contact with an analytic workspace's metadata. Therefore, you must define one metadata locator object for every analytic workspace whose data you wish to display in a Java client. Once you have defined the metadata locator object, be sure to assign the ISECMLOCATOR property to it —

if you fail to do so, OLAP Services will not be able to recognize that the analytic workspace contains metadata, and the OLAP API will be unable to display any data in that workspace.

Use the DEFINE command to create a text variable named ECMLOCATOR.

For example, use the following command to define a metadata locator object:

```
define ECMLOCATOR variable text
```

Properties required by the metadata locator object

After you define ECMLOCATOR, you must then set the following properties on it:

- ISECMLOCATOR
- DBDIMDIM
- DBMEASDIM
- DBATTRDIM
- DBFOLDERDIM

Setting the ISECMLOCATOR property

After you define ECMLOCATOR, then use the PROPERTY command to set the ISECMLOCATOR property to YES. Note that the name of any property must be enclosed in single quotes.

If the ECMLOCATOR variable is not the current object, then use the CONSIDER command before the PROPERTY command.

For example, use the following commands to set the ISECMLOCATOR property on the ECMLOCATOR variable:

```
consider ECMLOCATOR
property 'ISECMLOCATOR' yes
```

When the OLAP API opens an analytic workspace, an attempt is made to locate an OLAP DML object that has an ISECMLOCATOR property that is set to YES. This is how the OLAP API determines that the analytic workspace contains data that is described by analytic workspace metadata.

Set the DBDIMDIM property to identify the workspace's dimensions

DBDIMDIM is a property that specifies the name of the dimension dimension, which you must define and populate. This is a dimension whose dimension values are the names of the OLAP DML dimensions that exist in the analytic workspace.

To set the DBDIMDIM property, follow these steps:

1. Define and populate the dimension dimension.
2. Set the required properties on the dimension dimension.
3. Set the DBDIMDIM property on ECMLOCATOR. The DBDIMDIM property will specify the name of the dimension dimension that you have defined.

Defining the dimension dimension

Use the DEFINE DIMENSION command to define the dimension dimension. Then use the MAINTAIN command to specify the names of the dimensions that exist in the analytic workspace as the dimension values.

Suppose your analytic workspace contains three dimensions, named TIME, GEOGRAPHY, and PRODUCT. Use the following commands to define and populate a dimension named dbdimdim:

```
define dbdimdim dimension text
maintain dbdimdim add 'TIME' 'GEOGRAPHY' 'PRODUCT'
```

Setting properties on the dimension dimension

Once you have defined a dimension dimension, you then assign the following properties to it:

- LDSCVAR — Specifies the name of a text variable that contains the long descriptive labels of the dimensions. You must define this text variable, which should be dimensioned by the dimension dimension. See "Set the LDSCVAR property to identify an object's long description" on page A-25 for more information.
- NUMHIERFRM — Specifies the name of the formula that provides the number of hierarchies that are associated with each dimension. You only need to define one formula per analytic workspace. However, you must define this formula, and the formula must return the number of hierarchies for each dimension in the analytic workspace. See "Define a NUMHIERFRM object to determine the number of hierarchies" on page A-12 for more information.
- SDSCVAR — Specifies the name of a text variable that contains the short descriptive labels of the dimensions. You must define this text variable, which should be dimensioned by the dimension dimension. See "Set the SDSCVAR property to identify an object's short description" on page A-24 for more information.

For each property, define the object and populate it. You can then set the property on the dimension dimension.

For example, suppose you define and populate a variable named dbdim_ldscvar, a formula named dbdim_numhierfrm, and a variable named dbdim_sdscvar. Use the following commands to set the properties on a dimension dimension named dbdimdim:

```
consider dbdimdim
property 'LDSCVAR' 'dbdim_ldscvar'
property 'NUMHIERFRM' 'dbdim_numhierfrm'
property 'SDSCVAR' 'dbdim_sdscvar'
```

Setting the DBDIMDIM property on ECMLOCATOR

Once you have defined a dimension dimension, set the DBDIMDIM property on ECMLOCATOR. The following example specifies dbdimdim as the name of the dimension dimension:

```
consider eclocator
property 'DBDIMDIM' 'dbdimdim'
```

Set the DBMEASDIM property to identify the workspace's measures

DBMEASDIM is a property that specifies the name of a measure dimension, which you must define and populate. This is a dimension whose dimension values are the names of the OLAP DML variables that you have defined in the analytic workspace.

To set the DBMEASDIM property, follow these steps:

1. Define and populate the measure dimension.
2. Set the required properties on the measure dimension.
3. Set the DBMEASDIM property on ECMLOCATOR. The DBMEASDIM property will specify the name of the measure dimension that you have defined.

Defining the measure dimension

Use the DEFINE DIMENSION command to define the measure dimension. Then use the MAINTAIN command to specify the names of the variables that exist in the analytic workspace as the dimension values.

Suppose your analytic workspace contains three variables, named SALES, UNITS, and PROJECTED_SALES. You can use the following commands to define and populate a measure dimension named dbmeasdim:

```
define dbmeasdim dimension text
maintain dbmeasdim add 'SALES' 'UNITS' 'PROJECTED_SALES'
```

Setting properties on the measure dimension

The only property that you need to set for the measure dimension is the LDSCVAR property. See "Set the DBDIMDIM property to identify the workspace's dimensions" on page A-4 for an example of setting the LDSCVAR property.

Once you have defined a measure dimension, you then assign the following property to it:

LDSCVAR — Specifies the name of a text variable that contains the long descriptive labels of the variables. You must define this text variable, which should be dimensioned by the dimension dimension.

First, define the object and populate it. You can then set the property on the measure dimension.

For example, suppose you define and populate a variable named dbmeas_ldscvar. Use the following commands to set the property on a dimension dimension named dbdimdim:

```
consider dbdimdim
property 'LDSCVAR' 'dbmeas_ldscvar'
```

Setting the DBMEASDIM property on ECMLOCATOR

Once you have defined a measure dimension, set the DBMEASDIM property on ECMLOCATOR. The following example specifies dbmeasdim as the name of the measure dimension:

```
consider ECMLOCATOR
property 'DBMEASDIM' 'dbmeasdim'
```

Set the DBATTRDIM property to identify the workspace's attributes

DBATTRDIM is a property that specifies the name of an attribute dimension, which you must define and populate. This is a dimension whose dimension values are the names of the OLAP DML attributes that exist in the analytic workspace.

To set the DBATTRDIM property, follow these steps:

1. Define and populate the attribute dimension.
2. Set the required properties on the attribute dimension.
3. Set the DBATTRDIM property on ECMLOCATOR. The DBATTRDIM property will specify the name of the attribute dimension that you have defined.

For more information about attributes, refer to "Metadata That Describes Dimension Attributes" on page A-20.

Defining the attribute dimension

Use the DEFINE DIMENSION command to define the attribute dimension. Then use the MAINTAIN command to specify the names of the attributes that exist in the analytic workspace as the dimension values.

Suppose your analytic workspace contains three attributes, named USE, GENDER, and AGE. Use the following commands to define and populate a dimension named dbattrdim:

```
define dbattrdim dimension text
maintain dbattrdim add 'USE' 'GENDER' 'AGE'
```

Setting properties on the attribute dimension

Once you have defined an attribute dimension, you then assign the following properties to it:

- DOMAINDIMREL — Specifies the name of the relation that holds the dimension that functions as the domain of each attribute. See "Set the DOMAINDIMREL property on the attribute dimension" on page A-21 for more information.
- LDSCVAR — Specifies the name of the variable that contains the long descriptive labels of the attributes. See "Set the LDSCVAR property to identify an object's long description" on page A-25 for more information.
- RANGEDIMREL — Specifies the name of the relation that holds the dimension that functions as the range of each attribute. See "Set the RANGEDIMREL property" on page A-21 for more information.

For each property, define the object and populate it. You can then set the property on the attribute dimension.

For example, suppose you define and populate a relation named domaindimrel, a variable named dbattr_ldscvar, and a relation named rangedimrel. Use the following commands to set the properties for an attribute dimension named dbattrdim:

```
consider dbattrdim
property 'DOMAINDIMREL' 'domaindimrel'
property 'LDSCVAR' 'dbattr_ldscvar'
property 'RANGEDIMREL' 'rangedimrel'
```

For more information about an attribute's domain, an attribute's range, and setting properties for attributes, refer to "Metadata That Describes Dimension Attributes" on page A-20.

Setting the DBATTRDIM property on ECMLOCATOR

Once you have defined an attribute dimension, set the DBATTRDIM property on ECMLOCATOR. The following example specifies dbattrdim as the name of the attribute dimension:

```
consider ECMLOCATOR
property 'DBATTRDIM' 'dbattrdim'
```

Set the DBFOLDERDIM property to identify the workspace's folders

DBFOLDERDIM is a property that specifies the name of a folder dimension, which you must define and populate. This is a dimension whose dimension values are the names of the folders that exist in the analytic workspace. A folder is a collection of measures that a user considers to be related to one another.

For example, an analytic workspace might contain folders named Financial, Shipments, Market Measurements, Sales, and Human Resources. Folders typically have the same dimensionality, but this is not required. A measure can be in more than one folder. Folders can contain other folders.

To set the DBFOLDERDIM property, follow these steps:

1. Define and populate the folder dimension.
2. Set the required properties on the folder dimension.
3. Set the DBFOLDERDIM property on ECMLOCATOR. The DBFOLDERDIM property will specify the name of the folder dimension that you have defined.

Defining the folder dimension

Use the DEFINE DIMENSION command to define the folder dimension. Then use the MAINTAIN command to specify the names of the folders that exist in the analytic workspace as the dimension values.

Suppose your analytic workspace contains five folders, named FINANCIAL, SHIPMENTS, MARKET_MEASUREMENTS, SALES, and HUMAN_RESOURCES. You can use the following commands to define and populate a dimension named dbfolderdim:

```
define dbfolderdim dimension text
maintain dbfolderdim add 'FINANCIAL' 'SHIPMENTS' 'MARKET_MEASUREMENTS' -
   'SALES' 'HUMAN_RESOURCES'
```

Setting properties on the folder dimension

Once you have defined a folder dimension, you then assign the following properties to it:

- LDSCVAR — Specifies the name of the variable that contains the long descriptive labels of the folders. See "Set the LDSCVAR property to identify an object's long description" on page A-25 for more information.
- MEASINFOLDERVS — Specifies the name of the valueset that specifies the measures that belong to each folder. See "Set the MEASINFOLDERVS property to identify measures in folders" on page A-23 for more information.

For each property, define the object and populate it. You can then set the property on the folder dimension.

For example, suppose you define and populate a variable named dbfolder_ldscvar and a valueset named measinfoldervs. Use the following commands to set the properties for an folder dimension named dbfolderdim:

```
consider dbfolderdim
property 'LDSCVAR' 'dbfolder_ldscvar'
property 'MEASINFOLDERVS' 'measinfoldervs'
```

Setting the DBFOLDERDIM property on ECMLOCATOR

Once you have defined a folder dimension, set the DBFOLDERDIM property on ECMLOCATOR. The following example specifies dbfolderdim as the name of the folder dimension:

```
consider ECMLOCATOR
property 'DBFOLDERDIM' 'dbfolderdim'
```

Metadata That Describes Dimension Hierarchies

Metadata that is required for the display of dimension hierarchies

If a dimension has one or more hierarchies, and you want to make it possible to display those hierarchies on a Java client, then the following metadata is required for that dimension:

- "Set the HIERDIM property if a dimension has any hierarchies" on page A-11
- "Define a NUMHIERFRM object to determine the number of hierarchies" on page A-12

If you also want to display information about hierarchy levels, refer to "Metadata That Describes Dimension Hierarchy Levels" on page A-16.

Set the HIERDIM property if a dimension has any hierarchies

Define a hierarchy dimension to hold the name of every hierarchy in a dimension, then set the HIERDIM property on that dimension.

Defining the hierarchy dimension

Use the following steps to define a hierarchy dimension:

1. Define a dimension to act as the hierarchy dimension. The dimension values are the names of the hierarchies in a dimension. You must define a separate hierarchy dimension for every dimension that has hierarchies.
2. Create a self-relation for the dimension. For example, create a dimension that relates the TIME dimension to itself.
3. Limit the self-relation to each hierarchy.
4. Relate the dimension values in the self-relation.

For a detailed example of creating a hierarchy dimension, refer to the RELATION command in the OLAP DML Reference.

Setting the HIERDIM property on its dimension

After you have created a hierarchy dimension, then you can set the HIERDIM property on its dimension.

For example, suppose you create a hierarchy dimension for the TIME dimension named timehierdim. Use the following command to set the HIERDIM property for the TIME dimension.

```
consider time
property 'HIERDIM' 'timehierdim'
```

Define a NUMHIERFRM object to determine the number of hierarchies

Define one number-of-hierarchies formula whose return value is the number of that dimension's hierarchies. Because the formula is dimensioned by the dimension dimension, you only need to define one formula for each analytic workspace.

Defining the number-of-hierarchies formula

The number-of-hierarchies formula must have an integer data type and must be dimensioned by the dimension dimension. See "Set the DBDIMDIM property to identify the workspace's dimensions" on page A-4 for more information about the dimension dimension.

The following example defines a number-of-hierarchies formula.

```
define numhiers formula integer <dbdimdim>
eq if not obj(hasproperty 'HIERDIM' dimexpobj) then 0
   else if not exists(obj(property 'HIERDIM' dimexpobj)) then 0
   else obj(dimmax obj(property 'HIERDIM' dimexpobj))
```

For information about the HIERDIM property, see "Set the HIERDIM property if a dimension has any hierarchies" on page A-11.

For an example of defining a DIMEXPOBJ object, see "Set the EXPOBJVAR property on the attribute dimension" on page A-22.

Setting the NUMHIERFRM property on the dimension dimension

After you have defined the number-of-hierarchies formula, then you can set the NUMHIERFRM property on the dimension dimension:

```
consider dbdimdim
property 'NUMHIERFRM' 'numhiers'
```

Using the number-of-hierarchies formula

The following command uses the number-of-hierarchies formula to determine which dimensions have hierarchies:

```
limit dbdimdim to numhiers gt 0
```

Set the HIERDEFAULT property to specify the default hierarchy

If a dimension has more than one hierarchy, set the HIERDEFAULT property on that dimension and specify the name of the hierarchy that you want to be used as the default hierarchy.

Setting the HIERDEFAULT property

For example, suppose that the TIME dimension has two hierarchies: the Calendar hierarchy and the Fiscal hierarchy. These names are specified in the hierarchy dimension; refer to "Set the HIERDIM property if a dimension has any hierarchies" on page A-11 for more information.

If you want to use the Fiscal hierarchy as the default hierarchy for TIME, then use the following commands:

```
consider time
property 'HIERDEFAULT' 'Fiscal'
```

For information about naming hierarchies, see "Set the HIERDIM property if a dimension has any hierarchies" on page A-11.

Set the HIERLDSCVAR property to display hierarchy descriptions

To display a text description for a dimension's hierarchies, define a text variable, which must be dimensioned by the hierarchy dimension. Set the HIERLDSCVAR property on the dimension to specify the name of the object you create.

Defining the hierarchy description variable

The following example defines a variable named time.ldsc for the TIME variable. Note that the variable is dimensioned by TIME's hierarchy dimension, NUMHIERTIME.

```
define time.ldsc variable text <numhiertime>
```

Setting the HIERLDSCVAR property

After you have defined a hierarchy description variable, then you can set the HIERLDSCVAR property on its dimension.

For example, the following commands set the HIERLDSCVAR property on the TIME dimension and specify the variable named time.ldsc:

```
consider time
property 'HIERLDSCVAR' 'time.ldsc'
```

Set the DRILLINFOFRM to display information about drill direction

If information about whether dimension values can be expanded or contracted ("drilled") is important, then you can define a formula, then specify the formula name when you set the DRILLINFOFRM property on the dimension.

Defining the hierarchy drill information formula

The following example defines a formula named time.drill for the TIME dimension. Note that the formula is dimensioned both by the TIME dimension and by its hierarchy dimension, NUMHIERTIME. TIME.PARENT is the parent relation of the TIME dimension.

```
define time.drill formula integer <time numhiertime>
EQ if statlen(limit(TIME to children using TIME.PARENT
convert(TIME, int))) eq 0 then 0 else if statlen(TIME) eq
statlen(limit(TIME remove children using TIME.PARENT
convert(TIME, int))) then 1 else 2
```

The possible return values are:

- 0 — this dimension value has no children in the TIME hierarchy
- 1 — this dimension value has children and is not expanded
- 2 — this dimension value has children and is already expanded

Setting the DRILLINFOFRM property

After you have defined the hierarchy drill information formula, then you can set the DRILLINFOFRM property on the dimension.

For example, the following code sets the DRILLINFOFRM property on the TIME dimension and specifies the formula named time.drill:

```
consider time
property 'DRILLINFOFRM' 'time.drill'
```

Set the PARENTREL property to identify the hierarchy parent relation

The hierarchy parent relation is a self-relation in which you specify the parent of each dimension value. For example, suppose the TIME dimension values are months, quarters, and years. In the Calendar hierarchy, Q1 is the parent of January, February, and March. Therefore, you define a relation that relates TIME to itself, then relate Q1 as the parent of January, February, and March.

Once you have defined and populated the hierarchy parent relation, then set the PARENTREL property on that dimension.

Defining the hierarchy parent relation

The following example defines a self-relation for the TIME dimension. Note that the TIME dimension is related not only to itself but also to its hierarchy dimension, numhiertime.

```
define time.parent relation time <time, numhiertime>
```

You relate the dimension values (for example, for the Calendar year dimension values) in the self-relation (time.time) just as you would for a self-relation that has only one hierarchy. See the DEFINE RELATION command in the OLAP DML Reference for details.

Setting the PARENTREL property

After you have defined the hierarchy parent relation, then you can set the PARENTREL property on the dimension.

For example, the following code sets the PARENTREL property on the TIME dimension and specifies the relation named time.parent:

```
consider time
property 'PARENTREL' 'time.parent'
```

Set the FULLORDER property to identify the hierarchy organization

If a dimension has one or more hierarchies, define a hierarchy full order variable to identify the order of dimension values within each hierarchy, then set the FULLORDER property on that dimension.

Defining the hierarchy full order variable

The following example defines a variable for the TIME dimension. Note that the variable is defined with a decimal data type, and it is dimensioned by TIME and its hierarchy dimension, numhiertime.

```
define time.fullorder variable decimal <time, numhiertime>
```

You then assign a number for each combination of a TIME dimension value and a hierarchy name as the variable data.

Setting the FULLORDER property

After you have defined the hierarchy full order variable, then you can set the FULLORDER property on the dimension.

For example, the following code sets the FULLORDER property on the TIME dimension and specifies the variable named time.fullorder:

```
consider time
property 'FULLORDER' 'time.fullorder'
```

Metadata That Describes Dimension Hierarchy Levels

Metadata that is required for the display of hierarchy levels

If a dimension has one or more hierarchies, and you want to display the various levels of those hierarchies, then the following metadata is required for that dimension:

- "Set the LEVELDIM property to identify the names of hierarchy levels" on page A-16
- "Set the LEVELREL property to identify hierarchy level contents" on page A-17
- "Set the HIERLEVELVS property to specify level order" on page A-18
- "Set the LEVELDEPTHVAR property to identify hierarchy level depth" on page A-18
- "Set the LEVELLDSC property to identify hierarchy level descriptions" on page A-19

Set the LEVELDIM property to identify the names of hierarchy levels

Define a hierarchy level dimension to identify the name of each hierarchy level, then set the LEVELDIM property on the base dimension (meaning, the dimension whose hierarchy levels are being identified).

Defining the hierarchy level dimension

The following example defines a hierarchy level dimension for the TIME dimension. Note that the dimension is defined with a text data type.

```
define time.leveldim dimension text
```

You then add dimension values that are the names of the levels of the TIME dimension's hierarchies.

Setting the LEVELDIM property

After you have defined the hierarchy level dimension, then you can set the LEVELDIM property on the dimension.

For example, the following code sets the LEVELDIM property on the TIME dimension and specifies the dimension named time.leveldim:

```
consider time
property 'LEVELDIM' 'time.leveldim'
```

Set the LEVELREL property to identify hierarchy level contents

Define a dimension member level relation to identify the hierarchy level to which each dimension value belongs, then set the LEVELREL property on that dimension.

Defining the dimension member level relation

The following example defines a relation for the TIME dimension. Note that the relation relates the hierarchy level dimension for TIME, named time.leveldim to the TIME dimension and its hierarchy dimension, named time.hierdim. See "Set the LEVELDIM property to identify the names of hierarchy levels" on page A-16 for information about defining a hierarchy level dimension. See "Set the HIERDIM property if a dimension has any hierarchies" on page A-11 for information about defining a hierarchy dimension.

```
define time.levelrel relation time.leveldim <time, time.hierdim>
```

You then relate each dimension value to the hierarchy level to which it belongs. For information about populating a relation, refer to the DEFINE RELATION command in the OLAP DML Reference.

Setting the LEVELREL property

After you have defined the dimension member level relation, then you can set the LEVELREL property on the dimension.

For example, the following code sets the LEVELREL property on the TIME dimension and specifies the relation named time.levelrel:

```
consider time
property 'LEVELREL' 'time.levelrel'
```

Set the HIERLEVELVS property to specify level order

Define a level-to-hierarchy mapping valueset to hold the list of levels for each hierarchy, then set the HIERLEVELVS property on that dimension.

Order the levels from highest (meaning, the root level) to lowest. A level can belong to more than one hierarchy.

Defining the level-to-hierarchy mapping valueset

The following example defines a level-to-hierarchy mapping valueset for the TIME dimension. Note that the valueset is defined with a short integer data type. The definition also requires the level dimension and hierarchy dimension. For information about defining a level dimension, see "Set the LEVELDIM property to identify the names of hierarchy levels" on page A-16. For information about defining a hierarchy dimension, see "Set the HIERDIM property if a dimension has any hierarchies" on page A-11.

```
define time.hierlvl valueset time.leveldim <time.hierdim>
```

You then populate the valueset with a list of levels for each hierarchy. For information about populating valuesets, refer to the DEFINE VALUESET command in the OLAP DML Reference.

Setting the HIERLEVELVS property

After you have defined the level-to-hierarchy mapping valueset, then you can set the HIERLEVELVS property on the dimension.

For example, the following code sets the HIERLEVELVS property on the TIME dimension and specifies the valueset named time.hierlvl:

```
consider time
property 'HIERLEVELVS' 'time.hierlvl'
```

Set the LEVELDEPTHVAR property to identify hierarchy level depth

Define a level depth variable to identify each hierarchy level's distance from the hierarchy root, then set the LEVELDEPTHVAR property on that dimension.

The top level of the hierarchy has a value of 0.

Defining the level depth variable

The following example defines a level depth variable for the TIME dimension. Note that the variable is defined with a short integer data type. The variable is

dimensioned by TIME's level dimension and hierarchy dimension. For information about defining a level dimension, see "Set the LEVELDIM property to identify the names of hierarchy levels" on page A-16. For information about defining a hierarchy dimension, see "Set the HIERDIM property if a dimension has any hierarchies" on page A-11.

```
define time.lvldepth variable shortinteger <time.leveldim, time.hierdim>
```

You then add assign an integer for each hierarchy level that identifies its depth in the hierarchy.

Setting the LEVELDEPTHVAR property

After you have defined the level depth variable, then you can set the LEVELDEPTHVAR property on the dimension.

For example, the following code sets the LEVELDEPTHVAR property on the TIME dimension and specifies the variable named time.lvldepth:

```
consider time
property 'LEVELDEPTH' 'time.lvldepth'
```

Set the LEVELLDSC property to identify hierarchy level descriptions

Define a level long description variable to hold the description of each hierarchy level, then set the LEVELLDSC property on that dimension.

Defining the level long description variable

The following example defines a level long description variable for the TIME dimension. Note that the variable is defined with a text data type. It is dimensioned by TIME's hierarchy level dimension, named time.leveldim. For information about defining a hierarchy level dimension, see "Set the LEVELDIM property to identify the names of hierarchy levels" on page A-16.

```
define time.lvlldsc variable text <time.leveldim>
```

You then add text that describes the names of the levels of the TIME dimension's hierarchies.

Setting the LEVELLDSC property

After you have defined the level long description variable, then you can set the LEVELLDSC property on the dimension.

For example, the following code sets the LEVELLDSC property on the TIME dimension and specifies the variable named time.lvlldsc:

```
consider time
property 'LEVELLDSC' 'time.lvlldsc'
```

Metadata That Describes Dimension Attributes

What are attributes?

Attributes relate the dimension values in one OLAP DML dimension to the dimension values in another OLAP DML dimension.

An attribute has a domain dimension and a range dimension.

The domain dimension is the dimension about whose dimension values an attribute supplies information.

The range dimension values provide information about the domain dimension.

For example, suppose you define a PRODUCT dimension. One attribute is the color of each product. Therefore, you define a COLOR dimension to store the names of colors. PRODUCT is the domain dimension, and COLOR is the range dimension.

Any OLAP DML object can represent more than one analytic workspace attribute. Each unique mapping of domain and range dimensions to the dimensions of the object is a separate attribute.

Metadata that is required for the display of attributes

If a dimension has attributes and you want to display them, then the following metadata is required:

- "Set the DBATTRDIM property on the ECMLOCATOR object" on page A-20
- "Set the DOMAINDIMREL property on the attribute dimension" on page A-21
- "Set the RANGEDIMREL property" on page A-21
- "Set the EXPOBJVAR property on the attribute dimension" on page A-22

Set the DBATTRDIM property on the ECMLOCATOR object

You must set the DBATTRIDIM property on the ECMLOCATOR object in order to make it possible for attributes that exist in the analytic workspace to be found.

For details, refer to "Set the DBATTRDIM property to identify the workspace's attributes" on page A-7.

Set the DOMAINDIMREL property on the attribute dimension

Define a relation that holds the dimension that acts as domain of an attribute, then set the DOMAINDIMREL property on the attribute dimension.

Defining the attribute domain relation

The following example defines an attribute domain relation that relates the attribute dimension to the dimension dimension. For information about defining the dimension dimension, see "Set the DBDIMDIM property to identify the workspace's dimensions" on page A-4. For information about defining the attribute dimension, see "Set the DBATTRDIM property to identify the workspace's attributes" on page A-7.

```
define attrdomain relation dbdimdim <dbattrdim>
```

The dimension dimension, dbdimdim, contains the names of the dimensions in the analytic workspace. The attribute dimension, dbattrdim, contains the names of the dimensions that you have defined to store attribute values. You then store the name of the domain dimension in the relation. For information about defining and using relations, refer to the DEFINE RELATION command in the OLAP DML Reference.

Setting the DOMAINDIMREL property

After you have defined the attribute domain relation, then you can set the DOMAINDIMREL property on the attribute dimension.

For example, the following code sets the DOMAINDIMREL property on the DBATTRDIM dimension and specifies the relation named ATTRDOMAIN:

```
consider dbattrdim
property 'DOMAINDIMREL' 'attrdomain'
```

Set the RANGEDIMREL property

Define a relation that holds the dimension that acts as range of an attribute, then set the RANGEDIMREL property on the attribute dimension.

Defining the attribute range relation

The following example defines an attribute range relation that relates the attribute dimension to the dimension dimension. For information about defining the

dimension dimension, see "Set the DBDIMDIM property to identify the workspace's dimensions" on page A-4. For information about defining the attribute dimension, see "Set the DBATTRDIM property to identify the workspace's attributes" on page A-7.

```
define attrrange relation dbdimdim <dbattrdim>
```

The dimension dimension, dbdimdim, contains the names of the dimensions in the analytic workspace. The attribute dimension, dbattrdim, contains the names of the dimensions that you have defined to store attribute values. You then store the name of the range dimension in the relation. For information about defining and using relations, refer to the DEFINE RELATION command in the OLAP DML Reference.

Setting the RANGEDIMREL property

After you have defined the attribute range relation, then you can set the RANGEDIMREL property on the attribute dimension.

For example, the following code sets the RANGEDIMREL property on the DBATTRDIM dimension and specifies the relation named ATTRDOMAIN:

```
consider dbattrdim
property 'DOMAINDIMREL' 'attrrange'
```

Set the EXPOBJVAR property on the attribute dimension

Define a variable that holds the names of the attributes (which are typically relations) in the analytic workspace, then set the EXPOBJVAR property on the attribute dimension.

Defining the attribute name variable

An attribute is usually a relation between a dimension (such as PRODUCT) and a second dimension that holds descriptions (such as COLOR). For example, the second dimension holds names of colors that describe products.

The following is an example of a definition of a relation between COLOR and PRODUCT:

```
define color.product relation color <product>
```

The next example defines a variable that will store the names of the attributes in the analytic workspace. The variable has a text data type and is dimensioned by the attribute dimension. For information about defining the attribute dimension, see

"Set the DBDIMDIM property to identify the workspace's dimensions" on page A-4.

```
define attrexpobj variable text <dbattrdim>
```

You add the names of the attributes as the data to the variable and populate them. For example, after adding the name PRODUCTCOLOR to the ATTREXPOBJ variable, you can then assign the relation to it, as shown in the following example:

```
attrexpobj(dbattrdim 'PRODUCTCOLOR') = 'color.product'
```

Setting the EXPOBJVAR property

After you have defined the attribute name variable, then you can set the EXPOBJVAR property on the attribute dimension.

For example, the following code sets the EXPOBJVAR property on the DBATTRDIM dimension and specifies the variable named ATTREXPOBJ:

```
consider dbattrdim
property 'EXPOBJVAR' 'attrexpobj'
```

Metadata That Describes Other Objects

Metadata that is required for other OLAP DML objects

The following metadata is required for the display of the following objects:

- Folders — To display the measures in a business area, see"Set the MEASINFOLDERVS property to identify measures in folders" on page A-23.
- Any OLAP DML object — To provide a short description or a long description for any OLAP DML object, see "Set the SDSCVAR property to identify an object's short description" on page A-24 and "Set the LDSCVAR property to identify an object's long description" on page A-25.

Set the MEASINFOLDERVS property to identify measures in folders

Define a folder membership valueset to specify the measures that are contained in a folder, then set the MEASINFOLDERVS property on the folder dimension.

Defining the folder membership valueset

The following example defines a folder membership valueset. Note that the definition uses the measure dimension and is dimensioned by the folder dimension.

For information about the measure dimension, see "Set the DBMEASDIM property to identify the workspace's measures" on page A-6. For information about the folder dimension, see "Set the DBFOLDERDIM property to identify the workspace's folders" on page A-9.

```
define measinfldr valueset dbmeasdim <dbfolderdim>
```

You then create a list of measure names in the valueset. For more information about using valuesets, refer to the DEFINE VALUESET command in the OLAP DML Reference.

Setting the MEASINFOLDERVS property

After you have defined the folder membership valueset, then you can set the MEASINFOLDERVS property on the folder dimension.

For example, the following code sets the MEASINFOLDERVS property on the folder dimension and specifies the valueset named dbmeasdim.

```
consider dbfolderdim
property 'MEASINFOLDERVS' 'dbmeasdim'
```

Set the SDSCVAR property to identify an object's short description

To display a short name for any OLAP DML object, define a short description variable to hold the short name, then set the SDSCVAR property on that object.

Defining the short description variable

The following example defines a short description variable for a SALES variable, then assigns "sales" as the variable data. Note that the variable is defined with a text data type.

```
define short_sales variable text
short_sales = 'sales'
```

Setting the SDSCVAR property

After you have defined the short description variable, then you can set the SDSCVAR property on the OLAP DML object.

For example, the following code sets the SDSCVAR property on the SALES variable and specifies the short description variable named short_sales.

```
consider sales
property 'SDSCVAR' 'short_sales'
```

Set the LDSCVAR property to identify an object's long description

To display a long name for any OLAP DML object, define a long description variable to hold the long name, then set the LDSCVAR property on that object.

Defining the long description variable

The following example defines a long description variable for a SALES variable, then assigns "sales for the past 12 months" as the variable data. Note that the variable is defined with a text data type.

```
define long_sales variable text
long_sales = 'sales for the past 12 months'
```

Setting the LDSCVAR property

After you have defined the long description variable, then you can set the LDSCVAR property on the OLAP DML object.

For example, the following code sets the LDSCVAR property on the SALES variable and specifies the long description variable named long_sales.

```
consider sales
property 'LDSCVAR' 'long_sales'
```

Glossary

ad hoc analysis

A type of analysis in which you answer questions by manipulating the dimensions, dimension values, and layout of data. You can rotate the data in order to change its dimensional orientation. You can also drill down or up on designated values in order to expand or collapse dimension hierarchies.

(See also *dimension*; *dimension value*; *drill*; *hierarchy*; *variable*.)

administrator

See *database administrator (DBA)*.

aggregation

The consolidation of data for several dimension values into a single value, such as the total units sold for all the cities, or into a smaller set of values, such as the average units sold for cities in each region. Data is often collected at the lowest level of detail and is *aggregated* into higher level totals for analysis.

(See also *dimension*; *level*.)

ampersand substitution

An ampersand character (&) at the beginning of an expression tells OLAP Services to substitute the value of the expression for the expression itself in a command or function. This is useful in commands or functions that can take the name of an object as an argument as explained in "Substitution Expressions" on page 4-39.

See *aggregation*; *dimension value*; *hierarchy*; *level*; *object*; *parent*.

analytic workspace

A single file containing objects that organize and store data in a form that OLAP Services can use. You determine the structure and contents of an analytic workspace by defining objects, examples of which are dimensions, variables, and programs. Once these definitions are in the analytic workspace dictionary, you can enter, change, or use the data with OLAP Services.

You can use several analytic workspaces at the same time during an OLAP Services session. All the analytic workspaces attached to the session at the same time are active analytic workspaces. One of these analytic workspaces, the first one on the active analytic workspace list, is the current analytic workspace.

An analytic workspace usually consists of a single file, but you can have a *multifile analytic workspace* by specifying that the analytic workspace should be broken up into several files. This allows you to keep the files of a large analytic workspace at a manageable size.

(See also *current analytic workspace*; *dictionary*.)

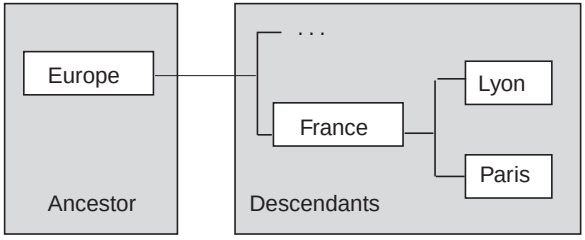
analytic workspace dictionary

See *dictionary*.

ancestor

(1) A dimension value at any level above a particular value in a hierarchy. The ancestor value is the aggregated total of the values of its descendants.

The following example shows the dimension value “Europe”” as the ancestor of “France,” “Lyon,” and “Paris” in the GEOGRAPHY dimension.



(2) If an application has an inheritance hierarchy, an ancestor is also an object that is two or more levels above a derived object. The level immediately above the derived object is the *parent*.

application

A set of OLAP DML objects plus external programs and files that work together to provide a unified collection of functions to the user. The collection of functions is designed to solve a user's problem.

The OLAP DML objects in an application can be in one analytic workspace or they can be in several different analytic workspaces. The characteristic that unifies them is their contribution to the application's purpose. For example, the purpose of one application might be to provide accounting assistance, while the purpose of another application might be to help with marketing analysis.

You can use a given object in more than one application. For example, a company's marketing application can use some of the same variables as its accounting application. In this case, the analytic workspace that holds the shared variables would be part of both applications.

argument

A keyword, expression, or object name that provides input to a command, function, method, or program. An argument can indicate the data values on which the command, function, method, or program operates. It can also specify the way in which the command, function, method, or program operates.

This term is relevant to the OLAP DML.

Another word for argument is parameter.

(See also *command*; *expression*; *function*; *object*.)

array

The format for storing data in analytic workspaces. You can also think of an array as a group of data cells arranged by the dimensions of the data. For example, a two-dimensional array is similar to a spreadsheet. The cells are arranged in rows and columns, with one dimension of the data forming the rows and the other forming the columns.

You can view a three-dimensional array as a group of data cells arranged in a cube, with each dimension forming one side of the cube. In the OLAP DML, arrays can have up to thirty-two dimensions.

Frequently, the term *array* is used as a synonym for a data variable, because a data variable is stored in an array.

attribute

A descriptive characteristic that is shared by dimension values. Attributes represent logical groupings that allow users to select data based on like characteristics. For example, in an analytic workspace representing footwear, you can use a shoe color attribute to select all boots, sneakers, and slippers that share the same color.

A keyword or phrase used in the DEFINE command to specify characteristics of an object definition in an analytic workspace. For example, the type of object, its data type, and its dimensions are all attributes specified in the DEFINE command.

An attribute is also a keyword or phrase used in commands that produce output (such as ROW or REPORT) to control the format of the output. For example, number of decimal places, column width, and centering are format attributes specified in ROW.

An attribute is also a keyword used in the FILESET command or the FILEQUERY function to specify or obtain the characteristics of a file unit. For information about file units, see the topic for FILEOPEN in the OLAP DML Reference.

(See also *dimension value*.)

AUTOGO

A program that you can write and that will be run automatically when the analytic workspace in which it resides is opened. An AUTOGO program is useful when you always want to execute a certain sequence of commands at start up. The AUTOGO program can execute any OLAP DML command, or run any of your own programs. For more information on AUTOGO, see the topic for the DATABASE command in the OLAP DML Reference.

base dimension

A dimension that is a component of a composite or a conjoint dimension. Composites and conjoints are used to control the size of data variables whose data is sparse. The values of a composite or conjoint are combinations of values from its base dimensions. A base dimension can be either a simple dimension or a conjoint dimension.

(See also *composite*; *conjoint dimension*; *dimension*; *sparsity*.)

basic data types

One of five general categories of data. When you define an OLAP DML variable or dimension, you give it a specific data type. The specific data types fall into the five basic data types: numeric, text, Boolean, date, and time.

Basic data types are not part of the definition of an OLAP DML object. However, it is sometimes useful to use these categories when explaining how the OLAP DML operates. Certain commands and functions treat all the data types in a basic category the same way; and a specific command or function might only apply to data types falling within one but not the other basic categories. For more information about the basic data types, see "OLAP DML Data Types" on page 4-2.

See *data type*.

Boolean expression

A logical formula that is either true or false for each value of the expression. For example, when you have the Boolean expression

```
actual gt 20000
```

each value of the variable ACTUAL is compared to the literal 20,000. If the value is greater than 20,000, then the formula is true; if the value is less than or equal to 20,000, then the formula is false.

For more information see "Boolean Expressions" on page 4-28.

cell

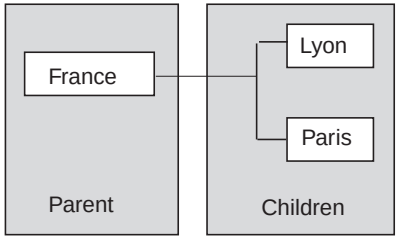
A single data value of an expression. In a dimensioned expression, a cell is identified by one value from each of the dimensions of the expression. For example, if you have a variable with the dimensions MONTH and DISTRICT, then each combination of a month and a district identifies a separate cell of that variable.

(See also *dimension*; *expression*; *variable*.)

child

(1) A dimension value at the level immediately below a particular value in a hierarchy. Values of children are included in the calculation that produces the aggregated total for a parent. A dimension value may be a child for more than one parent when the dimension has more than one hierarchy.

The following example shows the dimension values “Lyon” and “Paris” as the children of “France” in the GEOGRAPHY dimension.



(2) If your application has an inheritance hierarchy, a child is also an object that is derived from another object. The source object is called the *parent*.

(Contrast with *parent*. See also *aggregation*; *dimension value*; *hierarchy*; *level*; *object*.)

command

A word or group of words that instructs OLAP Services to start or stop an operation. Typically, a command consists of the command name followed by one or more arguments that specify the values on which the command is to operate and / or the conditions under which it is to operate.

(See also *argument*.)

compilable object

An OLAP DML object, such as a program, model, or formula, whose definition contains commands or expressions that are interpreted and executed each time you use the object. The commands and expressions that are included in the object are called the *source code*. By compiling the source code, OLAP Services can create *compiled code* that executes more quickly. A compilable object is automatically compiled the first time you use it after entering or changing the source code or when you use the COMPILE command. When you update the analytic workspace, the compiled code is saved as part of the analytic workspace and can be used later.

composite

A 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. A given combination is an index into one or more sparse data variables. The purpose for using a composite is to store sparse data in a compact form.

You are not required to specify the list of dimension-value combinations that will be included in your composite. Instead, values are added automatically to the list, based on the data in the variables that use the composite. A composite is not a dimension, but it is treated like one for the purpose of storing sparse data.

A composite can be named or unnamed. A named composite is an OLAP DML object that you have explicitly defined. An unnamed composite is automatically created when you define a variable with some dimensions specified as sparse. In this case, the composite is not an OLAP DML object.

(Contrast with *conjoint dimension*. See also *dimension*; *dimension value*; *sparsity*; *variable*.)

conjoint dimension

A dimension that you build on base dimensions. Each value in a conjoint is a combination of values, one from each of the conjoint's base dimensions. The purpose for using a conjoint is to achieve fine control over the status of individual combinations of base dimension values.

For storing sparse data, you should almost always use a composite instead of a conjoint, because composites are easier to use. The exception to this guideline is the case in which you want to be able to specify every dimension-value combination in status. In this situation, use a conjoint.

For a variable that is not sparse and not dimensioned by a conjoint, you might want to define a separate conjoint to hold a set of dimension-value combinations that meet one or more specific criteria. For example, you might include only dimension-value combinations for which the variable has values higher than a given number.

(Contrast with *composite*. See also *dimension*; *dimension value*; *sparsity*; *variable*.)

continuation character

A character indicating that a command continues on the next line. When you are editing a command in a program, model, or formula, and it will not fit on a single line, you can continue the command or response on additional lines. To do so, type a hyphen (-) or an equal sign (=) as the last character on the line. This specifies that the line is not complete but is continued on the next line.

current analytic workspace

The first analytic workspace on the active analytic workspace list (unless you do not have a current analytic workspace). You can display the dictionary of the current analytic workspace. You can modify and refer to objects, modify data, and run

programs in any analytic workspace attached to an OLAP Services session. For more information see Chapter 2.

See *analytic workspace*, *dictionary*.

current outfile

The current destination for the output of commands, such as REPORT and DESCRIBE, that produce text. If you have not used the OUTFILE command to send output to a file, then uses your default outfile is used. For more information see "Directing Output" on page 8-23.

See *default outfile*.

data-reading commands

A group of commands used in programs to read data from external files having different formats. These commands include FILEREAD, FILENEXT, FILEVIEW, FILEERROR, and RECNO. You use these commands with file I/O commands, such as FILEOPEN, FILECLOSE, FILEQUERY, FILESET, FILEGET, and FILEPUT. For more information see Chapter 11.

See *file I/O commands*.

data type

The kind of information contained in a variable or dimension (for example, whole numbers, decimal numbers, alphabetic characters, logical data). Each variable or dimension can have only one type of data. The OLAP DML data types are described in detail in "OLAP DML Data Types" on page 4-2.

database administrator (DBA)

The person responsible for creating, installing, configuring, and maintaining the analytic workspaces, so that users can access and analyze data effectively.

(See also *analytic workspace*.)

date literal

A sequence of characters used as a single DATE value in OLAP DML commands or functions. Normally, you must enclose a date literal in single quotes, so it will not be used as a number or as the name of a variable. A date literal includes components that identify the date of the day, month, and year. You can input date literals in styles such as `'24 April 2001'`, `'24/4/01'`, and `'240401'`. For a description of the input styles for dates and an explanation of how ambiguous input (such as

'030401') is interpreted, see the topic for the DATEORDER option in the OLAP DML Reference.

Normally, you can use a date literal whenever you are asked to specify a DATE expression in OLAP DML commands or functions.

For more information see "OLAP DML Data Types" on page 4-2.

DBA

See *database administrator (DBA)*.

decimal escape

A way of indicating a character using its EBCDIC or ASCII decimal value. The decimal escape for a character takes the following form.

\dnnn

The *d* indicates a decimal escape, and *nnn* is the decimal value for the character. Usually, the decimal escape must be enclosed in single quotes when used in OLAP DML commands.

If you specify an invalid decimal escape (that is, a decimal escape that does not represent an EBCDIC or ASCII character), then the decimal escape is converted directly to text. For example, the decimal escape '\d299' would be converted to the text value 'd299' because there is no EBCDIC or ASCII character with the decimal value 299.

For more information see "Text Expressions" on page 4-27, and see *escape sequence*, *hexadecimal escape*.

default outfile

The default destination for the output of commands, such as REPORT and DESCRIBE, that produce text. If you have not used the OUTFILE command to send output to a file, then your default outfile is used.

definition

A description of an OLAP DML object. Definitions are used to keep track of the information available in the analytic workspace. The collection of definitions in an analytic workspace is known as the *dictionary*.

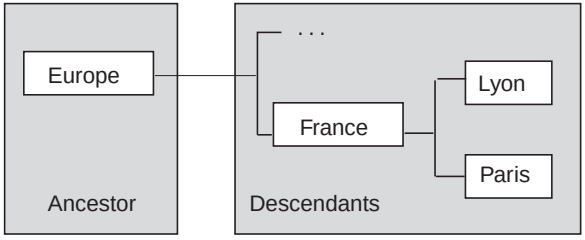
The definition includes the name of the object, its type (for example, DIMENSION or VARIABLE), its data type when applicable (for example, INTEGER or TEXT), and its dimensions. A definition might also include a description (LD), value name format (VNF) for a time dimension, an expression associated with a FORMULA,

permission specified with PERMIT commands, properties specified with the PROPERTY command, or the contents of a program or model. You can view one or more definitions in the analytic workspace with the DESCRIBE command. You can view properties with the FULLDSC command.

descendant

(1) A dimension value at any level below a particular value in a hierarchy. Values of descendants are included in the calculation that produces the aggregated total for an ancestor.

The following example shows the dimension values “France,” “Lyon,” and “Paris” as descendants of “Europe” in the GEOGRAPHY dimension.



(2) If an application has an inheritance hierarchy, a descendant is also an object two or more levels below another object. The level immediately below is the *child*.

(Contrast with *ancestor*. See also *aggregation*; *child*; *dimension value*; *hierarchy*; *level*; *object*.)

dictionary

The collection of definitions of the objects in an analytic workspace. The dictionary is also called the *analytic workspace dictionary*.

(See also *analytic workspace*; *dictionary*; *object*.)

dimension

A type of OLAP DML object that is a list of values that provide categories for data. A dimension acts as an index for identifying values of a variable. 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. A dimension is similar to a key in a relational database.

Any item of data within a multidimensional variable can be uniquely and completely selected by specifying one member from each of the variable’s

dimensions. For example, when a sales variable is dimensioned by MONTH, PRODUCT, and MARKET, specifying January for the MONTH dimension, Stereos for the PRODUCT dimension, and Eastern Region for the MARKET dimension uniquely specifies a single cell in the variable. Thus, dimensions offer a concise and intuitive way of organizing and selecting data for retrieval, updating, and performing calculations.

A dimension can be simple, with values that are single text or integer values, or it can be conjoint, with values that are combinations of values in other dimensions. A composite is not a dimension, but it is a conjoint-like internal object that is treated like a dimension for the purpose of handling sparse data.

(See also*composite*; *conjoint dimension*; *dimension value*; *multidimensional data*; *object*; *sparsity*; *variable*.)

dimension hierarchy

See *hierarchy*.

dimension label

A text description for a dimension. For example, a dimension that is named GEOGRAPHY might have the label “Geographic Areas”. The label, rather than the name, can be displayed in reports, tables, and graphs.

(See also *dimension*.)

dimension value

One element in the list that makes up a dimension. For example, a computer company might have dimension values in the PRODUCT dimension called LAPPC and DESKPC. Values in the GEOGRAPHY dimension might include Boston and Paris. Values in the TIME dimension might include MAY96 and JAN97.

See *dimension*.

dimension value label

A text description for a dimension value. For example, in a PRODUCT dimension that has values called LAPPC and DESKPC, the LAPPC value might have a label “Laptop PC”. Dimension value labels might appear as row, column, and page labels in reports or tables and as tick labels in graphs.

(See also *dimension*.)

DML

A data manipulation language (DML). In the OLAP Services environment, DML refers to the OLAP DML.

drill

To navigate up and down through the levels of aggregation in a dimension that has a hierarchy. When selecting dimension values or viewing data, you can expand or collapse a dimension hierarchy by drilling down or up in it. Drilling down expands the view to include child values that are associated with parent values in the dimension hierarchy. Drilling up collapses the list of descendant values associated with a parent value in the dimension hierarchy.

In a typical OLAP application, you can click on a plus icon to drill down on (expand) the hierarchy or on a minus icon to drill up on (collapse) the hierarchy.

(See also *child*; *dimension*; *hierarchy*; *level*; *parent*.)

EIF file

An EIF file is specially formatted for transferring data between analytic workspaces. You create an EIF file using the EXPORT command and read an EIF file using the IMPORT command.

embedded total

A predefined level of aggregation built into a dimension for which a hierarchy exists. For example, in a TIME dimension, each quarter represents the total for the months in the quarter. Data for embedded totals is calculated in the an analytic workspace rather than in an application.

(See also *aggregation*; *dimension*; *hierarchy*.)

escape sequence

A series of characters beginning with a backslash (\) to indicate special treatment by OLAP Services. The backslash is the escape character in the OLAP DML. It means that the characters that follow it should not be treated in the normal way. For example, when you tried to include an apostrophe in a TEXT value, as in the following expression,

```
'First Quarter's Earnings'
```

The second quote (in `Quarter's`) is interpreted as the end of the value. You can turn off this normal meaning of the quote by typing a backslash in front of it.

```
'First Quarter\'s Earnings'
```

The backslash is also used in a few cases to indicate special treatment of characters that normally do not have a special meaning. For example, the newline escape sequence (\n) can be used in TEXT literals to indicate a line break, although the character n does not normally have any special meaning.

Many OLAP DML commands take a file identifier as an argument. Path names for both DOS and UNC files generally require backslashes. For information on specifying file identifiers, see *file identifier* and *text literal* and "Text Expressions" on page 4-27.

expression

One or more data values that are specified in an OLAP DML program. For example, you can specify expressions as arguments to programs or functions. An expression can be any of the following:

- A single, literal value (for example, 10)
- A variable or formula that contains one or more values
- A function that returns one or more values
- A calculation that combines values, such as variables, formulas, or functions, with arithmetic or Boolean operators

(See also *argument*; *dimension*; *formula*; *function*; *program*; *variable*.)

extension files

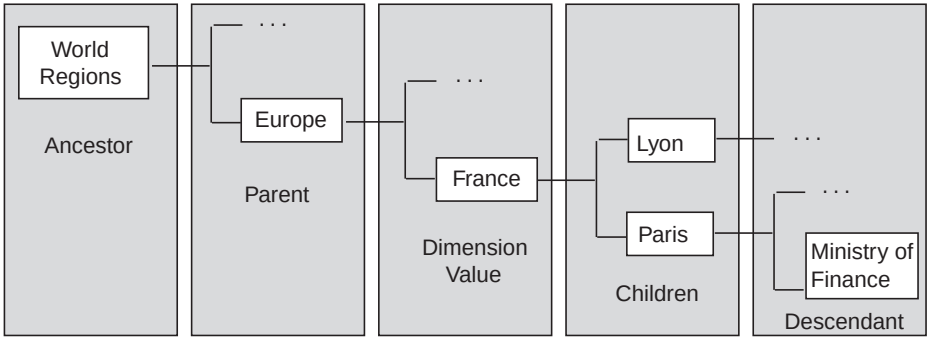
Let you divide a single analytic workspace among several files, so the analytic workspace can be larger than the space available on any single disk. Analytic workspace extension files turn a single analytic workspace into a *multifile analytic workspace*.

family

A group of related dimension values that correspond to the levels of the dimension hierarchy. Dimension values within a family can relate to each other as ancestor, parent, child, and descendant.

The following example shows family relationships for the dimension value "France" in the GEOGRAPHY dimension. The parent of "France" is "Europe,"

whose parent is “World Regions.” The children of “France” are “Lyon” and “Paris,” and one of the children of “Paris” is “Ministry of Finance.”



(See also *ancestor*; *child*; *descendant*; *dimension value*; *hierarchy*; *level*; *parent*.)

fastest-varying dimension

The first dimension listed in the definition of a variable or relation. When you are using a multidimensional variable or expression, the fastest-varying dimension is the one whose values vary first in a REPORT, =, or other command or function that loops over the dimensions of the expression.

For example, if you have a variable dimensioned by MONTH and CITY, then when you view the variable as REPORT command output, you will 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.

The order in which dimensions vary is determined by the way you defined the variable or relation being used. Dimensions vary in the order they are listed in the definition, with the first varying fastest and the last varying slowest.

When you use a variable as the solution variable in a model, the model will execute most efficiently when 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.

For more information see “Defining Variables” on page 3-11.

See *model*, *solution variable*.

file I/O commands

A group of input/output commands for handling external files on a detailed level. The file I/O commands let you open, close, and delete files; read and write lines of text; and query and set various file attributes. You use them in programs with data-reading commands to bring external data into an analytic workspace. For more information on the file I/O commands, see Chapter 11.

See *data-reading commands*.

file identifier

Many OLAP DML commands take a file identifier as an argument. In the Windows environment, the format in which you specify a file name depends on where the file is located. For files that are local to the computer on which OLAP Services is running, use DOS format. For files remote to the computer on which OLAP Services is running, use either DOS or UNC format, unless explicitly stated otherwise in the documentation. However, be consistent; all references to a given file must be in the same format.

- DOS file name format is as follows

  ```
  [d:] [\][path\] filename[.ext]
  ```

  where *d* designates a disk drive; a backslash (\) character may follow the drive name; *path* is a path of directory names separated by backslash (\) characters; *filename* is the name of the file, and *ext* is a 1- to 3-character extension preceded by a period.

- UNC file name format is as follows

  ```
  \\host\share\[path\] filename[.ext]
  ```

  where *host* designates the host system; *share* designates a shared area on the host; *path* is a path of directory names separated by backslash (\) characters; *filename* is the name of the file, and *ext* is a 1- to 3-character extension preceded by a period.

When specifying file identifiers in OLAP DML commands, it is good practice to always enclose them in single quotes. This will prevent parsing errors in cases where file name components are also OLAP DML object names or reserved words.

See *escape sequence*, *text literal*

file name

See *file identifier*.

fileunit

Any destination, such as a disk file, to which output can be sent. An arbitrary integer is assigned to a fileunit when it is opened in a session. A fileunit is opened with the OUTFILE command, which sets the current destination for command output, or with the FILEOPEN function. The default outfile is also a fileunit.

formula

A type of OLAP DML object that represents a stored calculation, expression, or procedure that produces a value. A formula provides a way to define and save complex or frequently used relationships within the data without resaving the data itself. Each time you use a formula, the calculation or procedure that is required to produce the value is performed.

(See also *expression*; *object*.)

function

A programming language routine that returns a value. You can use a function wherever an expression is required, by specifying the name of the function followed by its arguments enclosed in parentheses.

The OLAP DML includes built-in functions. In addition, it allows you to create functions of your own.

(See also *argument*; *expression*.)

hexadecimal escape

A way of indicating a character using its hexadecimal value. The hexadecimal escape for a character takes the following form.

\xnn

The *x* indicates a hexadecimal escape, and *nn* is the hexadecimal value for the character. Usually, the hexadecimal escape must be enclosed in single quotes when used in OLAP DML commands.

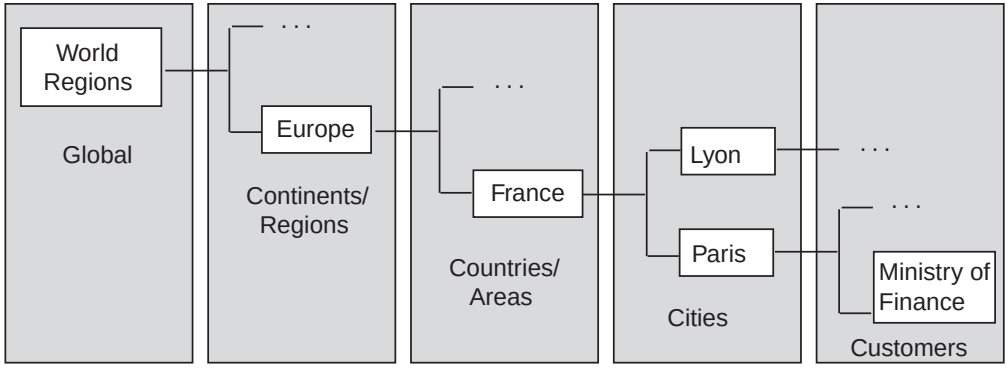
If you specify an invalid hexadecimal escape (that is, a hexadecimal escape that does not represent an EBCDIC or ASCII character), then the hexadecimal escape is converted directly to text. For example, the hexadecimal escape `'\xwhat'` would be converted to the text value `'xwhat'`.

(See also *escape sequence*; *decimal escape*.)

hierarchy

A means of organizing and structuring data. 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 based on them.

The following example shows a hierarchy based on the GEOGRAPHY dimension, in which dimension values are arranged in five levels. Data at the Customers level is aggregated into the Cities level, which, in turn, is aggregated into the Countries / Areas, Continents / Regions, and Global levels.



(See also *aggregation*; *ancestor*; *child*; *descendant*; *dimension value*; *level*; *parent*.)

input file

A disk file containing one or more OLAP DML commands. You can instruct that the input from this file be read by using the INFILE command. You can only have one command per line in an input file. However, with continuation characters, one command might occupy several lines in an input file.

Arrange the commands in the order in which you want them to be executed. The file must also include, in proper sequence, the appropriate responses to any prompts resulting from the commands.

label (in a program)

A string of characters followed by a colon and included as a separate line in a program. A label is used to mark the beginning of a section of a program that you want to execute under certain conditions. You can branch from one part of the program to a section headed by a label, altering the sequence in which commands

are executed or skipping some commands altogether. Commands such as GOTO and TRAP allow you to specify conditions and to branch to labels.

A label must start with a letter, dot, or underscore, and the remaining characters must be letters, numbers, dots, or underscores. Because only the first eight characters of a label name are used, you can experience problems with label names greater than eight characters, when the first eight characters are not unique. A label can contain up to 497 characters (the maximum length of a text line minus one character for the colon identifying a label).

LD (description)

A description attached with an LD command to an object in an analytic workspace. LDs are used primarily to document an analytic workspace by attaching an explanatory description to the objects defined in the analytic workspace. The LD is saved as part of the analytic workspace dictionary and is included when you describe an object. Through the OBJ function, you can use LDs to annotate output produced by report programs.

Normally, you use the LD command to supply the LD when you define an object. However, you can add or change an LD at any time with the CONSIDER and LD commands. You can attach a description to any type of object.

level

A position in a dimension hierarchy. Each level above the base level represents the aggregated total of the data from the level below. For example, the TIME dimension might have ascending levels such as Month, Quarter, and Year. Within a dimension hierarchy, a dimension value at one level has a family relationship with the dimension values at the levels above and below that level.

(See also *aggregation*; *dimension value*; *family*; *hierarchy*.)

local variable

A single-cell variable defined and used within an OLAP DML program. A local variable is not an OLAP DML object. When the program is finished executing, any local variables are erased. A local variable can have any data type that a variable object can have and can also be a relation to a dimension, meaning it holds a value of that dimension. A local variable is always a single value (although it can be a multiline text value); it does not have any dimensions.

(See also *multiline text value*; *single-cell variable*; *variable.*)

metadata

Data that describes other data. An example of metadata is a variable that lists the names of levels in a hierarchy or that holds the number of decimal places to be used for displaying data. Client applications use metadata when displaying multidimensional data in graphs, reports, tables, and so on.

(See also *multidimensional data*; *object*.)

model

A type of OLAP DML object that 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.

(See also *dimension value*; *object*; *variable.*)

multidimensional data

Data organized by two or more dimensions. With two dimensions, the data is structured as an array with rows and columns. With three dimensions, it is structured as a cube in which each dimension forms an edge. Structures with more than three dimensions have no physical metaphor, but they can organize data in ways that are useful for analysis.

Multidimensional analytic workspaces are optimized for complex data analysis. For example, a Sales variable might be dimensioned by TIME, PRODUCT, and GEOGRAPHY, so that only a few short steps would be needed to find the 10 cities with the top sales of tents over the last 3 months. In a relational database, a complex SQL program would be needed to get the same information.

(See also *cell*; *dimension*.)

multifile analytic workspace

An analytic workspace usually consists of a single file, but you can have a multifile analytic workspace by specifying that an analytic workspace be broken into several files. This allows you to keep the files of a large analytic workspace at a manageable size.

(See also *cell*; *dimension*.)

multiline text value

A TEXT value that occupies more than one line. A line can be up to 498 characters in length. If broken into lines, then a TEXT value can exceed the 498-character line limit. You can create a multiline TEXT value with the JOINLINES function.

You can also specify a multiline TEXT literal by including the escape sequence \n wherever you want a line break to occur. For example, the following literal value

```
'Yes\nNo'
```

is interpreted as the following two-line value.

```
Yes
No
```

NA value

A special data value that indicates that data is "not available" (NA). NA is the value of any cell to which a specific data value has not been assigned or for which data cannot be calculated.

(See also *cell*; *sparsity*.)

numeric literal

A sequence of digits used in OLAP DML commands or functions. A numeric literal can be preceded by a plus (+) or minus (-) sign and can contain a decimal point; however, it cannot contain commas.

Examples of numeric literals are:

```
1
25600
-471
.001
15.7
-6.342
```

Normally, you can use a numeric literal whenever you are asked to specify a numeric expression in OLAP DML commands or functions.

object

In the OLAP DML, a distinct item in the analytic workspace, which is defined as an entry in the analytic workspace dictionary. Objects are the basic pieces of an analytic workspace. When you build an analytic workspace, you must define one or more objects to organize, store, and retrieve the data. These objects include dimensions, variables, relations, formulas, and programs.

(See also *analytic workspace*; *dictionary*.)

OLAP

Online analytical processing. OLAP is a category of software technology that enables analysts, managers, and executives to gain insight into data by accessing a wide variety of views of information. Such information has been organized to reflect the real dimensionality of the user's enterprise.

OLAP functionality is characterized by dynamic, multidimensional analysis of consolidated enterprise data, which supports analytical and navigational activities such as the following:

- Calculating and modeling across dimensions and through hierarchies
- Analyzing trends over sequential time periods
- Creating slices of data for on-screen viewing
- Drilling down to lower levels of consolidation
- Reaching through to underlying detail data
- Rotating to change the dimensional orientation in the viewing area

OLAP analysis tools run against a multidimensional data engine or interact directly with a relational database management system (RDBMS).

(See also *dimension*; *drill*; *hierarchy*; *model*; *multidimensional data*;.)

option

A special type of OLAP DML object. Generally, an option allows you to control the format of output (for example, COMMAS, DECIMALS) or to turn on / off special OLAP DML operations (for example, PRGTRACE, DIVIDEBYZERO). You cannot define an option as part of an analytic workspace. However, you can use any of the options that are defined as part of the OLAP DML.

outfile

The destination for the output of commands, such as REPORT and DESCRIBE, that produce text.

(See also *current outfile*; *default outfile*.)

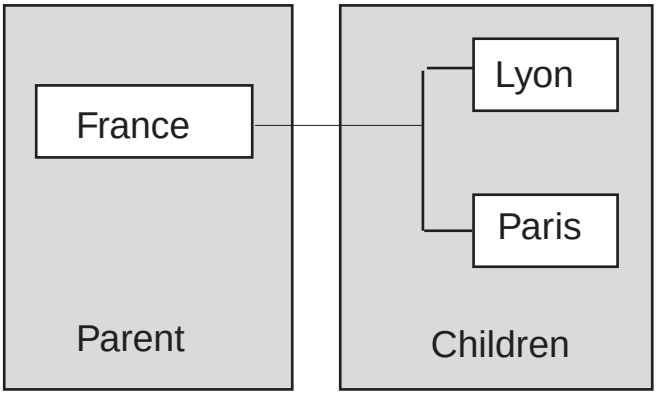
page

A unit of storage in an analytic workspace. A page can contain data or it can be a free page, meaning it is available for use when you add to or change the database. Use the DATABASE function to find the number of bytes in a page.

parent

(1) A dimension value at the level immediately above a particular value in a hierarchy. The parent value is the aggregated total of the values of its children.

The following example shows the dimension value “France” as the parent of “Lyon” and “Paris” in the GEOGRAPHY dimension.



(2) In an application that has an inheritance hierarchy, a parent is also an object from which one or more objects have been derived. Objects derived from the parent are called children.

(Contrast with *child*. See also *aggregation*; *dimension value*; *hierarchy*; *level*.)

permission

Restricted access to OLAP DML objects, as specified with PERMIT commands. You can use PERMIT commands in an analytic workspace security system that specifies access rights for many users. You can also use PERMIT as an application building tool for scoping.

(See also *scoping*; *status*.)

program

A type of OLAP DML object that contains a series of OLAP DML commands. A program is a stored procedure that executes a set of related commands. Programs can be nested, with one calling another to create a complete application or sophisticated analytic workspace maintenance tool. A program can return a value; in this case, it is called a user-defined function.

(See also *command*; *object*.)

property

In the OLAP DML language, a named value associated with an object. You can use properties to store information about objects, such as the number of decimal places to use when preparing reports on the object. You specify properties with the PROPERTY command.

Also, a characteristic of an object or component. Properties provide identifiers and descriptions, define object features (such as the number of decimal places or the color), or define object behaviors (such as whether an object is enabled).

(See also *object*.)

QDR

See *qualified data reference (QDR)*.

qualified data reference (QDR)

A qualifier that limits one or more of an expression's dimensions to a single value. A QDR is useful when you want to temporarily reference a value without affecting the current status. In the following example of an OLAP DML command, the QDR limits the MONTH dimension to "JUN95."

```
fetch sales(month 'JUN95')
```

Avoid using QDRs for complex expressions. Instead, use the QUAL function.

(See also *dimension*; *dimension value*; *expression*; *status*.)

relation

A type of OLAP DML object that establishes a correspondence between the values of a given dimension and the values of that dimension or other dimensions in the analytic workspace. For example, you might have a relation between cities and sales regions, such that each city belongs to a particular region.

A relation is similar to a single-dimensioned variable. However, it is distinct from most variables, because each cell holds the value of a dimension. For example, in a relation between cities and sales regions, the relation would be dimensioned by CITY. Each cell would hold the corresponding value of the REGION dimension.

(See also *cell*; *dimension*; *dimension value*; *variable*.)

report

A tabular presentation of multidimensional data. A report is an analysis tool that is used to view, manipulate, and print data.

(See also *reporting commands*.)

reporting commands

An informal collection of commands and options used to write report programs. These commands and options allow you to create custom reports, in which you can specify virtually any format for each row of the report and put headings and titles on the report pages. Other commands and functions allow you to calculate totals and to handle errors that might occur in the production of the report. The report writing commands are discussed in Chapter 12.

scalar

See *single-cell variable*.

scoping

Restricting the view of OLAP DML objects. You can use the PERMIT command to restrict access to values with read permission.

(See also *permission*.)

selection

The set of dimension values currently chosen for a dimension, or the script that contains conditions or criteria to specify those values.

(Contrast with *status*. See also *dimension value*.)

simple dimension

A dimension whose values are single elements with a data type of TEXT, ID, or INTEGER, or a time data type (DAY, WEEK, MONTH, QUARTER, or YEAR). For more information, see “Defining Dimensions” on page 3-4.

(See also *conjoint dimension*; *dimension*; *time dimension*.)

single-cell variable

A variable that has no dimensions. Since it has no dimensions, a single-cell variable has only one cell, which can contain a single data value. A single-cell variable can have any data type. If it is a TEXT variable, then it can have a single multiline value. A single-cell variable is sometimes called a scalar or a scalar variable.

slowest-varying dimension

The last dimension listed in the definition of a variable or relation. When you are using a multidimensional variable or expression, the slowest-varying dimension is the one whose values vary last in a REPORT, =, or other command or function that loops over the dimensions of the expression.

For example, if you have a variable dimensioned by MONTH and CITY, then when you view the variable as REPORT command output, you will see the data for all months for the first city before you see any data for the second city. In this case, CITY is the slowest-varying dimension because its values change after those of MONTH.

The order in which dimensions vary is determined by the definition of the variable or relations being used. Dimensions vary in the order you list them in the definition, with the first varying fastest and the last varying slowest.

solution variable

A variable that serves as both the source and the target of data in a model containing dimension-based equation. You specify the name of the solution variable when you run the model.

When you use a variable as the solution variable in a model, the model will execute most efficiently when 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.

(See also *model*.)

sparsity

A concept that refers to multidimensional data in which a relatively high percentage of the combinations of dimension values do not contain actual data. Such “empty,” or NA, values take up storage space in the analytic workspace. To handle sparse data efficiently, you can create a composite.

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 past months in the MONTH dimension, but the cells contain NA values.
- Random sparsity — Occurs when NA values are scattered throughout the 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 some of those products and other ones, too.

(See also *composite*; *multidimensional data*; *NA value*.)

status

The list of currently accessible values for a given dimension. If the status of a given dimension is limited to a subset of its stored values, then all expressions that are based on that dimension will be limited to the corresponding subset of data.

(Contrast with *selection*. See also *dimension*; *dimension value*; *expression*.)

text literal

A sequence of alphabetic characters used as a single TEXT or ID value in OLAP DML commands or functions. Normally, you must enclose a text literal in single quotes, so that it will not be used as the name of a variable. A text literal can contain any combination of alphabetic characters (including digits). Normally, you can use a text literal whenever you are asked to specify a text expression in OLAP DML commands or functions. For more information, see "OLAP DML Data Types" on page 4-2.

See *escape sequence*, *file identifier*.

time dimension

A dimension with a time data type. The values of a time dimension represent time periods that correspond to the data type of the dimension. For more information, see "Defining Dimensions" on page 3-4.

See *dimension*.

update

To save on disk any changes made to the analytic workspace during an OLAP Services session. Changes made during a session only affect the analytic workspace in working memory. To save these changes permanently on disk, you must issue an UPDATE command.

user-defined function

To augment the predefined functions, you can define a *program* that behaves like a function by returning a value.

(See also *function*.)

valueset

A type of OLAP DML object. A valueset contains a list of dimension values for a particular dimension. After defining a valueset, you use the LIMIT command to assign values from the dimension to the valueset. The values in a valueset can be saved across OLAP Services sessions.

When you begin a new OLAP Services session or open an analytic workspace, each dimension has all values in the status. You can then limit a dimension to the values stored in the valueset for that dimension. For more information, see "Working with Valuesets" on page 6-25.

(See also *dimension*; *status*.)

variable

A type of OLAP DML object that stores data. The data type of a variable indicates the kind of data that it contains.

If a variable has dimensions, then those dimensions organize its data, and there is one cell for each combination of dimension values. A dimensioned variable is an array whose cells are individual data values. If a variable has no dimensions, then it is a single-cell variable, which contains one data value.

(See also *array*; *cell*; *dimension*; *dimension value*; *object*.)

VNF (value name format)

A value name format attached with a VNF command to a time dimension in an analytic workspace. The VNF controls the input and output format for values of the dimension. It can include format specifications for any of the components that identify a time period (day, month, calendar year, fiscal year, and period within a fiscal year).

The VNF is saved as part of the analytic workspace dictionary and appears whenever you describe the time dimension. Normally, you use the VNF command to supply the VNF when you define the dimension. However, you can add or change a VNF at any time using the CONSIDER and VNF commands.

(See also *time dimension*.)

working memory

The temporary working area used by OLAP Services. OLAP Services does not work directly on the permanent disk copy of the data. Instead, when data is needed for calculations, reports, and so on, OLAP Services copies it from the disk file into the computer's virtual memory. When you examine, manipulate, or change data during

an OLAP Services session, you are affecting only the copy of the data in working memory.

Working memory gives you some protection for the data, and at the same time lets you try things out without worrying about what is happening to the data. You can make changes you are not sure you want to keep, just to see how they look.

When you leave OLAP Services, the copy of the analytic workspace in working memory is discarded, and the permanent disk copy remains unchanged unless you used the UPDATE command to copy the changes to disk.

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z