Oracle Warehouse Builder 11*g*
Getting Started

Extract, transform, and load data to build a dynamic, operational data warehouse

Bob Griesemer

PACKT PUBLISHING

BIRMINGHAM - MUMBAI

Oracle Warehouse Builder 11*g*: Getting Started

Copyright © 2009 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: August 2009

Production Reference: 1300709

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847195-74-6

`www.packtpub.com`

Cover Image by Parag Kadam (`paragvkadam@gmail.com`)

Download at Boykma.Com

Credits

Author
Bob Griesemer

Reviewers
Anitha Kadaru
Yasodarani Venkatesan

Acquisition Editor
James Lumsden

Development Editor
Swapna V. Verlekar

Technical Editors
Arani Roy
Reshma Sundaresan

Copy Editor
Sneha Kulkarni

Editorial Team Leader
Abhijeet Deobhakta

Project Team Leader
Lata Basantani

Project Coordinators
Ashwin Shetty
Neelkanth Mehta

Indexer
Rekha Nair

Proofreader
Chris Smith

Production Coordinator
Adline Swetha Jesuthas

Cover Work
Adline Swetha Jesuthas

Download at Boykma.Com

About the Author

Bob Griesemer has over 27 years of software and database engineering/DBA experience in both government and industry, solving database problems, designing and loading data warehouses, developing code, leading teams of developers, and satisfying customers. He has been working in various roles involving database development and administration with the Oracle Database with every release since Version 6 of the database from 1993 to the present. He has also been performing various tasks, including data warehouse design and implementation, administration, backup and recovery, development of Perl code for web-based database access, writing Java code utilizing JDBC, migrating legacy databases to Oracle, and developing Developer/2000 Oracle Forms applications. He is currently an Oracle Database Administrator Certified Associate, and is employed by the Northrop Grumman Corporation, where he is the Senior Database Engineer and primary data warehouse ETL specialist for a large data warehouse project.

I would like to thank my two co-workers, Anitha Kadaru and Yasodarani Venkatesan, who were kind enough to review this book. With their wealth of knowledge of data warehousing and Business Intelligence, they provided invaluable comments that helped me keep the book on track. I'd like to thank David Allan of the Oracle Warehouse Builder development team at Oracle for putting up with my numerous questions and requests for clarification about various aspects of the software. Lastly and most importantly, I'd like to thank my wife Lynn and children Robby, Melanie, Hilary, Christina, Millie, and Mikey for doing without a husband and dad for major periods of time over the last year, while I worked on this book. Your understanding and support has been a big help!

Download at Boykma.Com

About the Reviewers

Anitha Kadaru is employed with Northrop Grumman and has more than 12 years of experience in leading and supporting Information Technology (IT) development, including 10 years of experience of directly supporting the Decision Support Systems (DSS). She provides expertise in a broad range of Common Off-The-Shelf (COTS) applications for Business Intelligence (BI), data integration, and data architectures, and she is expert in all phases of system lifecycle development for the DSS applications. She has in-depth technical knowledge and exceptional analytical skills with implementing the COTS solutions in data warehousing, the ETL, and the BI technical areas. She has expertise in data engineering with years of data analysis, data design, dimensional modeling, and data management expertise.

Yasodarani Venkatesan is employed by Northrop Grumman as a Data Warehouse Analyst on a Healthcare project. In the past 11 years, she has worked on several large and small data warehousing projects in sales, logistics, finance, healthcare, and HR domain areas. She has expertise in designing and modeling star and snowflake schema design, designing and implementing the ETL processes for converting/transforming data, designing and implementing metadata layers in the Business Intelligence (BI) applications, and quality assurance.

Download at Boykma.Com

Download at Boykma.Com

Table of Contents

Download at Boykma.Com

Download at Boykma.Com

Download at Boykma.Com

Download at Boykma.Com

Preface

Competing in today's world requires a greater emphasis on strategy, long-range planning, and decision making, and this is why businesses are building data warehouses. Data warehouses are becoming more and more common as businesses have realized the need to mine the information that is stored in electronic form. Data warehouses provide valuable insight into the operation of a business and how best to improve it. Organizations need to monitor these processes, define policy, and at a more strategic level, define the visions and goals that will move the company forward in the future. If you are new to data warehousing in general, and to **Extract, Transform, and Load (ETL)** in particular, and need a way to get started, the Oracle Warehouse Builder is a great application to use to build your warehouse. The **Oracle Warehouse Builder** (**OWB**) is a tool provided by Oracle that can be used at every stage of the implementation of a data warehouse right from the initial design and creation of the table structure to ETL and data-quality auditing.

We will build a basic data warehouse using Oracle Warehouse Builder. It has the ability to support all phases of the implementation of a data warehouse from designing the source and target information, the mappings to map data from source to target, the transformations needed on the data, and building the code to implementing the mappings to load the data. You are free to use any or all of the features in your own implementation.

What this book covers

This book is an introduction to the **Oracle Warehouse Builder (OWB)**. This is an introductory, hands-on book so we will be including in this book the features available in Oracle Warehouse Builder that we will need to build our first data warehouse.

Download at Boykma.Com

The chapters are in chronological order to flow through the steps required to build a data warehouse. So if you are building your first data warehouse, it is a good idea to read through each chapter sequentially to gain maximum benefit from the book. Those who have already built a data warehouse and just need a refresher on some basics can skip around to whatever topic they need at that moment.

We'll use a fictional toy company, ACME Toys and Gizmos, to illustrate the concepts that will be presented throughout the book. This will provide some context to the information presented to help you apply the concepts to your own organization. We'll actually be constructing a simple data warehouse for the ACME Toys and Gizmos company. At the end of the book, we'll have all the code, scripts, and saved metadata that was used. So we can build a data warehouse for practice, or use it as a model for building another data warehouse.

Chapter 1: An Introduction to Oracle Warehouse Builder starts off with a high-level look at the architecture of OWB and the steps for installing it. It covers the schemas created in the database that are required by OWB, and touches upon some installation topics to provide some further clarification that is not necessarily found in the Oracle documentation. Most installation tasks can be found in the Oracle **README** files and installation documents, and so they won't be covered in depth in this book.

Chapter 2: Defining and Importing Source Data Structures covers the initial task of building a data warehouse from scratch, that is, determining what the source of the data will be. OWB needs to know the details about what the source data structures look like and where they are located in order to properly pull data from them using OWB. This chapter also covers how to define the source data structures using the Data Object Editor and how to import source structure information. It talks about three common sources of data—flat files, Oracle Databases, and Microsoft SQL Server databases—while discussing how to configure Oracle and OWB to connect to these sources.

Chapter 3: Designing the Target Structure explains designing of the data warehouse target. It covers some options for defining a data warehouse target structure using relational objects (star schemas and snowflake schemas) and dimensional objects (cubes and dimensions). Some of the pros and cons of the usage of these objects are also covered. It introduces the Warehouse Builder for designing and starts with the creation of a target user and module.

Download at Boykma.Com

Chapter 4: Creating the Target Structure in OWB implements the design of the target using the Warehouse Builder. It has step-by-step explanations for creating cubes and dimensions using the wizards provided by OWB.

Chapter 5: Extract, Transform, and Load Basics introduces the ETL process by explaining what it is and how to implement it in OWB. It discusses whether to use a staging table or not, and describes mappings and some of the main operators in OWB that can be used in mappings. It introduces the Warehouse Builder Mapping Editor, which is the interface for designing mappings.

Chapter 6: ETL: Putting it Together is about creating a new mapping using the Mapping Editor. A staging table is created with the Data Object Editor, and a mapping is created to map data directly from the source tables into the staging table. This chapter explains how to add and edit operators, and how to connect them together. It also discusses operator properties and how to modify them.

Chapter 7: ETL: Transformations and Other Operators expands on the concept of building a mapping by creating additional mappings to map data from the staging table into cube and dimensions. Additional operators are introduced for doing transformations of the data as it is loaded from source to target.

Chapter 8: Validating, Generating, Deploying, and Executing Objects covers in great detail the validating of mappings, the generation of the code for mappings and objects, and deploying the code to the target database. This chapter introduces the Control Center Service, which is the interface with the target database for controlling this process, and explains how to start and stop it. The mappings are then executed to actually load data from source to target. It also introduces the Control Center Manager, which is the user interface for interacting with the Control Center Service for deploying and executing objects.

Chapter 9: Extra Features covers some extra features provided in the Warehouse Builder that can be very useful for more advanced implementations as mappings get more numerous and complex. The metadata change management features of OWB are discussed for controlling changes to mappings and objects. This includes the recycle bin, cutting/copying and pasting objects to make copies or backups, the snapshot feature, and the metadata loader facility for exporting metadata to a file. Keeping objects synchronized as changes are made is discussed, and so is the auto-binding of tables to dimensional objects. Lastly, some additional online references are provided for further study and reference.

Download at Boykma.Com

What you need for this book

The following software is required for this book:

- Oracle Warehouse Builder 11*g*
- Microsoft SQL Server 2008 Express with Advanced Services

Who this book is for

If you are new to data warehousing and you have to build your first data warehouse using OWB, or have implemented a data warehouse using another tool and are now using OWB for the first time, this book is for you. You can also use it as a refresher if you are a more advanced user. An ever-increasing number of businesses are implementing data warehouses and if you are reading this book, then even yours has most likely chosen to implement one.

This book is for anyone tasked with building a data warehouse and loading data into it using Oracle Warehouse Builder. It is primarily aimed at database administrators and engineers who are new to data warehousing and are building a data warehouse for the first time using OWB. This book can also be used as a refresher on basic OWB features. Think of it as a beginner's guide to OWB. It can be helpful for any IT professional looking to broaden his or her knowledge about data warehousing in general and Oracle Warehouse Builder in particular.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Just substitute your applicable `ORACLE_HOME` location."

A block of code is set as follows:

```
# HS init parameters
#
HS_FDS_CONNECT_INFO = <odbc data_source_name>
HS_FDS_TRACE_LEVEL = <trace_level>
#
# Environment variables required for the non-Oracle system
#
#set <envvar>=<value>
```

Download at Boykma.Com

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
# HS init parameters
#
HS_FDS_CONNECT_INFO = <odbc data_source_name>
HS_FDS_TRACE_LEVEL = <trace_level>
#
# Environment variables required for the non-Oracle system
#
#set <envvar>=<value>
```

Any command-line input or output is written as follows:

```
@ORACLE_HOME\owb\rtp\sql\show_service.sql
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "We will click on the **Install** button to proceed with the installation."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an email to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book on, see our author guide on `www.packtpub.com/authors`.

Download at Boykma.Com

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code for the book

Visit `http://www.packtpub.com/files/code/5746_Code.zip` to directly download the example code.

The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration, and help us to improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to any list of existing errata. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or web site name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

Download at Boykma.Com

1
An Introduction to Oracle Warehouse Builder

The **Oracle Warehouse Builder** (**OWB**) is what this book is all about, so let's start discussing it by looking at it from a high level. We'll talk about some installation topics and the various components that compose this application. Oracle provides some detailed installation documentation and user guides that give you step-by-step instructions on how to install the product and the prerequisites we need to have in place. So we will focus more on some general topics that will help us understand the installation better. We'll walk through a basic installation that can be followed along and actually performed while reading. We'll be accepting most of the defaults during the installation for simplicity. For more advanced installation requirements, dig into the Oracle installation documentation to get familiar with the options that are available. You can find this at `http://www.oracle.com/pls/db111/homepage` by clicking on the **Installing and Upgrading** link in the lefthand frame.

Introduction to data warehousing

Although you may not be familiar with data warehousing, you have probably at least heard the term. Data warehouses are becoming increasingly common as businesses have realized the need to be able to mine the information they have stored in the electronic form in order to provide a valuable insight into the operation of their business and how best to improve it. Organizations need to monitor these processes, define policies, and—at a more strategic level—define the visions and goals that will move the company forward in the future. Operational transactional systems have greatly benefited the daily functioning of the enterprise. But now, organizations are shifting to a more decisional-based requirement from their computing platforms and are looking to build data warehouses. This is where OWB enters the picture to help organizations with the task of building that data warehouse.

Download at Boykma.Com

Introduction to our fictional organization

The manuals that Oracle supplies with its database and applications contain a great deal of information. However, it can be hard to relate that information to the real-world ways of implementing the database and applications. Anyone who has ever tried to read a technical user guide or reference provided with a database or application will know what that means. It is a great benefit to be able to learn about a new software tool by seeing how that tool is actually used within the context of an actual organization conducting a business. This is precisely the focus of this book. We'll be building an actual data warehouse using a fictional organization as an example.

Before we talk about what a data warehouse is, let's get introduced to the fictional organization we'll be using to demonstrate the use of the Warehouse Builder to build a data warehouse. Throughout this book, we will be using examples of the concepts involved by making reference to a fictional organization named *ACME Toys and Gizmos*, which is sales oriented. It is an entirely made-up organization, and any similarity to a real company is completely coincidental. This book will provide explanations throughout on how to use the OWB tool to build a data warehouse within the context of this invented company, which is involved in storefront and online Internet sales. Thus, it will demonstrate practical ways of implementing a data warehouse that can be directly applied in the real world.

ACME Toys and Gizmos will have stores all over the United States as well as a number of other countries, and will also have an online storefront for Internet sales. The **online transactional processing** systems **(OLTP)** play a huge role in the functioning of any business today, especially in the operation of a sales-oriented business. So this makes a good example to illustrate the subject matter of data warehousing and how to take information from those OLTP systems to load our warehouse.

Although we'll be using a sales organization for our examples, the concepts we'll discuss can apply to any business and will be as generic as possible to assist in doing that.

What is a data warehouse?

We've discussed the business case for implementing a data warehouse by showing how companies these days need information to support strategic-level decision making. We've also introduced the fictional organization that we'll use to provide examples of the concepts we'll be presenting. But we've not yet explained what a data warehouse is.

Download at Boykma.Com

We will not be dealing in detail with the concept of a data warehouse as that topic would encompass the entire contents of a book by itself. There are a number of good books already written about that topic. Therefore, we will touch upon some high-level concepts only as an introduction and to provide a context for using OWB to build a data warehouse.

Fundamentally, a data warehouse is a decisional database system. It is designed to support the decision makers in the organization in ways a transactional processing system is ill-equipped to handle, such as the strategic-level goals and visions of an organization. To think strategically, a large amount of data over long periods of time is needed. Transactional systems are concerned with the day-to-day operations such as: How many dolls did we sell today and will we need to restock the inventory? How many orders were processed today? How many balls were shipped out today? The strategic thinkers are more concerned with questions such as: How many dolls did we sell today compared to the same time period in the last year? How has our inventory level been for the last few months?

To support that level of information, we need more data than what is provided by the day-to-day transactions. We'll need much more information compiled over greater time periods and this is where the data warehouse comes in. As a data warehouse is different from a transactional database, there are some unique terms used to describe the data it contains. There are also other techniques that should be employed for designing the database for a data warehouse, which would not be a good idea for a transactional database.

The data in a data warehouse is composed of facts (actual numerical measures) and dimensions (descriptive data about those measures) that place the facts in a context that is understandable to the end-user decision maker. For instance, a customer makes a purchase of a toy with ACME Toys and Gizmos on a particular day over the Internet, which results in a dollar amount of the transaction. The dollar amount becomes the fact and the toy purchased, the customer, and the location of the purchase (the Internet in this case) become the dimensions that provide a scope of the fact measurement and give it a meaning.

The design of a data warehouse should be different from that of a transactional database. The data warehouse must handle large amounts of data, and must be simple to query and understand by the end users. While relational techniques and normalization are excellent database design methods for transactional systems to ensure data integrity, they can make understanding a data warehouse difficult for the end users. They can also bog down a data warehouse with long-running queries that have to make use of many joins (including more than one table that share a common data element to look up additional data).

Download at Boykma.Com

A much better means of representing the data is to de-normalize the data, so that users will not have to be concerned with retrieving the data from multiple tables. The use of foreign keys (a column that references a row in another table) should be restricted in a data warehouse. The outcome is a fact table with foreign keys only to each of the dimension tables. The diagram of the database structure has a fact table in the middle surrounded by dimension tables, resulting in something that looks like a star. Thus, the term star schema is used to refer to this representation of a data warehouse. It is also possible that these dimensions may themselves have other tables surrounding them, resulting in something akin to a snowflake. Thus, the term snowflake schema is also used. This is the dimensional modeling technique of representing a data warehouse.

This design lends itself extremely well to the task of querying large amounts of data by the end users. Users do not have to be bothered with queries involving complicated joins with multiple tables to get the descriptive information they need. This is because the information is included directly in the dimension tables in a de-normalized fashion. If a manager for ACME Toys and Gizmos needs to know what products sold well in the last quarter, the query will only involve two tables—the main fact table containing the data on number of items sold and the product dimension table that contains all the information about the product. The de-normalization means the manager will not have to be concerned with looking up product information in any other tables, as all the details about the product will be included in the one dimension table.

All this is great background information on data warehouses, but you can read any number of other books for much more detailed material on the topic. Our purpose in this book is to introduce the Oracle Warehouse Builder and use it to design and build our first data warehouse. So, let's see how it fits in to this discussion of data warehousing.

Where does OWB fit in?

The Oracle Warehouse Builder is a tool provided by Oracle, which can be used at every stage of the implementation of a data warehouse, from initial design and creation of the table structure to the ETL process and data-quality auditing. So, the answer to the question of where it fits in is—everywhere. It is provided as a part of the Oracle Database Release 11*g* installation. For the previous Oracle Database Releases, it can be downloaded and installed from Oracle's web site as a free download.

Download at Boykma.Com

We can choose to use any or all of the features as needed for our project, so we do not need to use every feature. Simple data warehouse implementations will use a subset of the features and as the data warehouse grows in complexity, the tool provides more features that can be implemented. It is flexible enough to provide us a number of options for implementing our data warehouse as we'll see in the remainder of the book.

Installation of the database and OWB

We'll be using the latest version of the database as of this writing—*Oracle Database 11g Release 1*—and the corresponding version of OWB that (as of this release) is included with the database install. If you have that version of the database installed already, you can skip this section and move right on to the next. If not, then keep reading as we discuss the installation of the database software.

Downloading the Oracle software

We can download the Oracle database software from Oracle's web site, provided we adhere to their license agreement. This agreement basically says we agree to use the database and the accompanying software either for development of a prototype of our application or for our own learning purposes. If we proceed to use this application internally or make it commercially available, then we will need to purchase a license from Oracle. For the purpose of working through the contents of this book to learn OWB, we need to install the database, which is covered under the license agreement for the free download.

We can find the database on the **Oracle Technology Network** web site (`http://www.oracle.com/technology`). The main database download is usually the first download listed under **FEATURED DOWNLOADS** on the main page. We need to register on the site, in order to create an account, before it lets us download any files, but there is no charge for that. The download files are classified by the platform on which they can be executed, so we'll choose the one for the system we'll be hosting the database on. We'll have to accept the license agreement first before the web page will let us download the file. The download files are anywhere from 1.7 GB to 2.3 GB in size, depending on the platform we'll be hosting it on. So we do not want to attempt this download unless we have a Broadband Internet connection (that is, cable, DSL, and so on). We'll download the install file and unzip it to a folder on a drive with enough available space. The installation files are temporary and are not needed after the installation is done, so we'll be able to delete them to free up space if needed.

Download at Boykma.Com

A word about hardware and operating systems

When installing software of this magnitude, we have to decide whether we'll have to buy additional hardware and a different operating system to run the database and OWB. OWB will run in the following databases:

- Oracle Database 11*g* R1 Standard Edition
- Oracle Database 11*g* R1 Enterprise Edition
- Oracle Database 10*g* R2 Standard Edition
- Oracle Database 10*g* R2 Enterprise Edition

This list is for the most recent version of OWB, which we'll be using throughout this book. We can download older versions of OWB that will run on older versions of the database, but we will not have the benefit of the improvements as in the latest version of the software. Much of what we'll be doing with the software throughout the course of the book can also be done on previous versions of the software. However, due to the changes made to things such as the interface, it would be easiest to follow along using the most recent version.

For this book, the platform is Windows Vista with Oracle Database 11*g* Release 1 (11.1.0.6) Enterprise Edition (which is the most recent version as of this writing), which is available from the download site. The Enterprise Edition of the database was chosen because it allows us to make full use of the features of the Warehouse Builder, especially in the area of dimensional modelling. There are some errors that will be generated by the client software when running in the Standard Edition installation due to code dependencies. These code dependencies are in libraries that are installed with the Enterprise Edition, but not the Standard Edition. We could use OWB with the Standard Edition, but then we would be limited in the type of objects we could deploy. For instance, dimensions and cubes would be problematic, and without using them we'd be missing out on a major functionality provided by the tool. If we want to develop any reasonably-sized data warehouse, the Enterprise Edition is the way to go.

Everything that we'll work through in this book was done on a laptop personal computer with an Intel Core 2 processor running at 1.67 GHz and 2 GB of RAM. Oracle says 1 GB of RAM will suffice, but it is always good to have more to provide better performance. Minimum specifications usually result in underpowered systems for all but the very basic processing. In terms of hard disk space, Oracle specifies that 4.5 GB is required for the basic database installation. We'll need about 2 GB just to save the installation files, so to make sure we have plenty of space, we should plan for something between 10 GB and 15 GB of available disk space just to be safe. We don't want to install the database software and then find that we don't have any space on our hard drive.

Download at Boykma.Com

Oracle supports its database installed on Windows and Unix. For Windows, it supports Windows XP Professional or Windows Vista (Business Edition, Enterprise Edition, or Ultimate Edition) as well as Windows Server 2003. The system mentioned above that was used for writing this book and working through all the examples, is running Windows Vista Home Premium Edition with Service Pack 1 and the database installed runs on it. We certainly would not want to use this configuration for large production databases, but it works fine for simple databases and learning purposes. The installation program will first do a prerequisite check of the computer and will flag any problems that it sees, such as not enough memory or an incorrect version of the operating system. For working through this book on our own to learn about the Warehouse Builder, we should be OK as long as we are running XP or Vista. However, for business users who would be installing the Oracle Database and OWB for use at work using Windows, it would be a good idea to stick with the recommended configurations of Windows XP Professional, Windows Vista (Business Edition, Enterprise Edition, and Ultimate Edition), or Windows Server.

Server versus workstation

We don't have to use a computer that is configured as a server to host the Oracle database. It will get installed on a regular workstation as long as the minimum system requirements are met. However, we might encounter a minor issue. A workstation is usually configured to use **Dynamic Host Configuration Protocol** (**DHCP**) to obtain its IP address. This means the address is not specified as a fixed address and can change the next time the system boots up. The Oracle database requires a fixed address to be assigned, and it can install on a system with DHCP. But it will also require the Microsoft Loopback Adapter to be installed as the primary network interface to provide that fixed address. If this situation is encountered, the installer prerequisite checks will alert us to that and give us instructions on how to proceed. It will not harm our existing network configuration to install that option. That is the way the laptop mentioned above was configured for this book project.

Installing Oracle database software

So far we've decided what system we're going to host the database on, downloaded the appropriate install file for that system, and unzipped the install files into a folder to begin the installation. We'll navigate to that folder and run the `setup.exe` file located there. This will launch the **Oracle Universal Installer** program to begin the installation.

Download at Boykma.Com

We are installing the full database, which now automatically includes the Warehouse Builder client and database components. If we had an older version of the database (10*g* R2 for example) that did not include the Warehouse Builder software, or if we wanted to run the client on a different workstation than where the database software is installed, then there is the option to install the Warehouse Builder by itself.

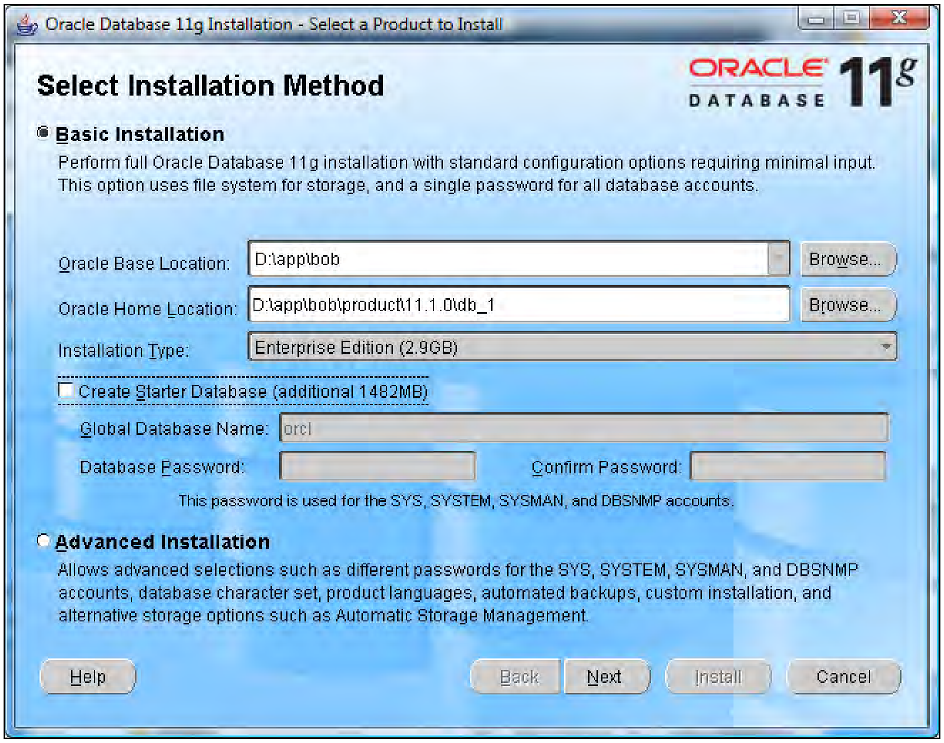
A separately downloadable install for the standalone option is available at `http://www.oracle.com/technology/software/products/warehouse/index.html`. Skip ahead to the section titled *Installing the OWB standalone software* if just the Warehouse Builder software is needed.

One of the first questions the installer will ask us is about setting up our **ORACLE_HOME**—the destination to install the software on the system and the name of the home location. Oracle uses this information when running to determine where to find its files on the system. It will store the information in the registry on Windows. It will suggest a default name, which can be changed. We'll leave it set to the default—`OraDb11g_home1`.

The **ORACLE_BASE** and **ORACLE_HOME** locations will have suggested paths filled in for us. It is a good idea to leave the path names as they are and only change the drive designation if we'd like to install to a different hard drive. The install program will suggest a drive for the installation, but we might have a different preference.

Oracle recommends a convention for naming folders and files that they call the **Optimal Flexible Architecture** (**OFA**). This is described in *Appendix B* of the *Oracle Database Installation Guide* for Microsoft Windows, which can be found at the following URL: `http://download.oracle.com/docs/cd/B28359_01/install.111/b32006/ofa.htm#CBBEDHEB`. It is a good idea to follow their recommendations for standardization so that others who have to work with the database files will know where to find them, and to save us from problems with possible conflicts with other Oracle products we may have installed. If we keep the default folder locations intact and only change the drive letter, we will adhere to the standard. We'll be asked to choose our installation method and whether to install a starter database. We're not going to let it install a starter database for us because it's going to default to a transactional database and we want a data warehouse. So on the **Select Installation Method** screen, we'll check off the **Basic Installation** type and uncheck the box for installing a starter database. The **Select Installation Method** screen should look similar to the following:

Download at Boykma.Com



Oracle Database 11g Installation - Select a Product to Install

Select Installation Method

(●) **Basic Installation**

Perform full Oracle Database 11g installation with standard configuration options requiring minimal input. This option uses file system for storage, and a single password for all database accounts.

Oracle Base Location: D:\app\bob

Oracle Home Location: D:\app\bob\product\11.1.0\db_1

Installation Type: Enterprise Edition (2.9GB)

[ ] Create Starter Database (additional 1482MB)

Global Database Name: orcl

Database Password:

Confirm Password:

This password is used for the SYS, SYSTEM, SYSMAN, and DBSNMP accounts.

( ) **Advanced Installation**

Allows advanced selections such as different passwords for the SYS, SYSTEM, SYSMAN, and DBSNMP accounts, database character set, product languages, automated backups, custom installation, and alternative storage options such as Automatic Storage Management.

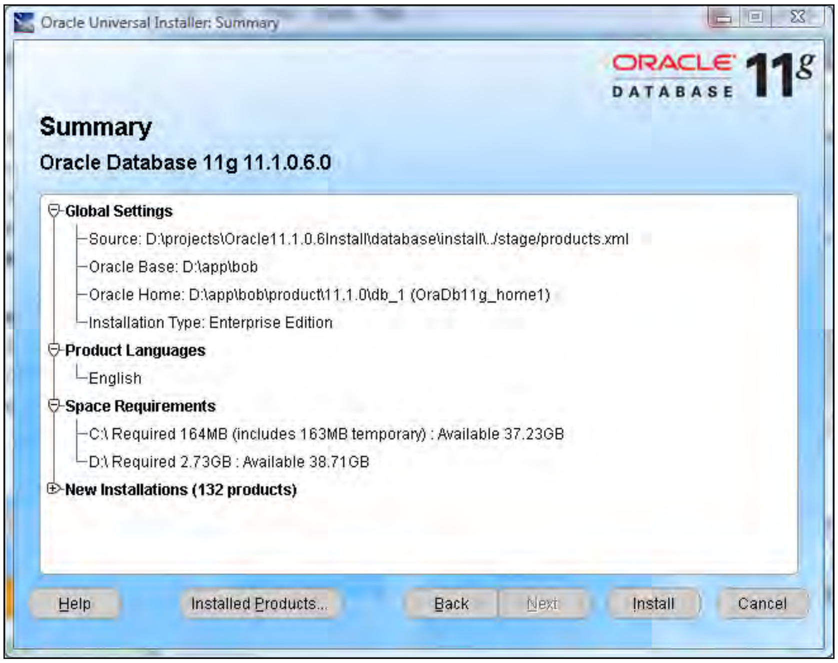
Basic versus advance install

The *Basic Installation* method is the quickest and easiest, but makes many decisions for us that the *Advanced Installation* option will ask us about. For the purpose of working through the examples in this book, we will be OK with the basic installation. But if we were installing for a production environment, we would want to read through the *Oracle Database Installation Guide* (`http://www.oracle.com/technology/documentation/database.html`; click on **View Library** to view the documentation online or click on **Download** to download the documentation) to familiarize ourselves with the various situations that would require us to use the advanced installation option. This would ensure that we don't end up with a database installation that will not support our needs.

We will click on the **Next** button to continue and the install program will perform its prerequisite checks to ensure our system is capable of running the database. That should show a status of **Succeeded** for all the checks. If any of the checks do not pass, we have to correct them and start over before continuing. When everything reports a status of **Succeeded**, we can click on the **Next** button.

Download at Boykma.Com

The install screens will proceed to the **Summary** screen where we can verify the options that we selected for the installation before actually doing it. So here we can make any last minute changes.



Summary

Oracle Database 11g 11.1.0.6.0

Global Settings
- Source: D:\projects\Oracle11.1.0.6Install\database\install\../stage/products.xml
- Oracle Base: D:\app\bob
- Oracle Home: D:\app\bob\product\11.1.0\db_1 (OraDb11g_home1)
- Installation Type: Enterprise Edition

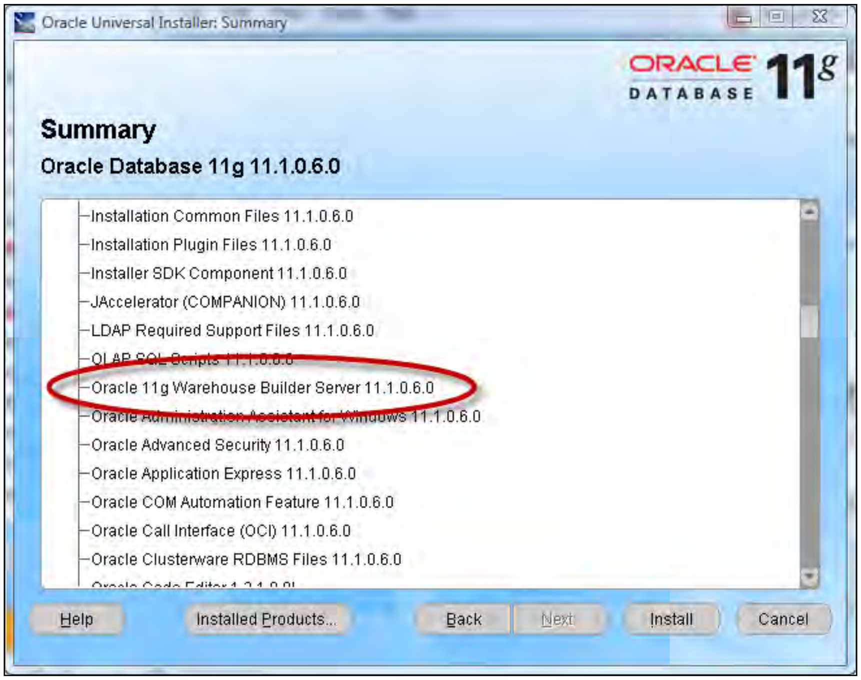
Product Languages
- English

Space Requirements
- C:\ Required 164MB (includes 163MB temporary) : Available 37.23GB
- D:\ Required 2.73GB : Available 38.71GB

New Installations (132 products)

If we expand the **New Installations** entry, it will list all of the database products and features that will be installed. This includes the feature we are most interested in, the **Oracle 11g Warehouse Builder Server** option, which is included automatically in 11*g* database installations. The following image illustrates what will appear in the list for OWB and the option we are interested in is circled:

Download at Boykma.Com



Summary

Oracle Database 11g 11.1.0.6.0

```
-Installation Common Files 11.1.0.6.0
-Installation Plugin Files 11.1.0.6.0
-Installer SDK Component 11.1.0.6.0
-JAccelerator (COMPANION) 11.1.0.6.0
-LDAP Required Support Files 11.1.0.6.0
-OLAP SQL Scripts 11.1.0.6.0
-Oracle 11g Warehouse Builder Server 11.1.0.6.0
-Oracle Administration Assistant for Windows 11.1.0.6.0
-Oracle Advanced Security 11.1.0.6.0
-Oracle Application Express 11.1.0.6.0
-Oracle COM Automation Feature 11.1.0.6.0
-Oracle Call Interface (OCI) 11.1.0.6.0
-Oracle Clusterware RDBMS Files 11.1.0.6.0
```

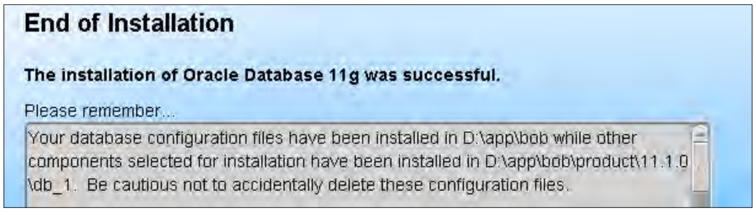
Now that we've specified our installation method and verified the options and components to be installed, we will click on the **Install** button to proceed with the installation. We'll be presented with the progress screen as it performs the installation with a progress bar that proceeds to the right as it installs.

Location of install results

A good idea is to pay particular attention to a message at the bottom of the install progress screen, which tells us where we can find a log of the installation. The logs that the installer keeps are stored in the Oracle folder on the system drive in the following subfolder: `C:\Program Files\Oracle\Inventory\logs`. The files are named with the following convention: `install ActionsYYYY-MM-DD_HH-MI-SSPM` where `YYYY` is the year, `MM` the month, `DD` the day, `HH` the hour, `MI` the minutes, `SS` the seconds of the time the installation was performed, and `PM` is either AM or PM. The files will have a `.log` extension. This information may come in useful later to see just what products were installed. The folder also will contain any errors encountered during the installation in files with a file extension of `.err` and any output generated by the installer in files with a file extension of `.out`.

Download at Boykma.Com

When it completes we'll be presented with the final screen with a little reminder similar to the following where bob is the login name of the user running the installation:


End of Installation

The installation of Oracle Database 11g was successful.

Please remember...

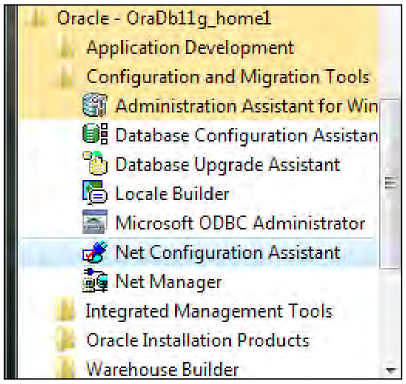
Your database configuration files have been installed in D:\app\bob while other components selected for installation have been installed in D:\app\bob\product\11.1.0\db_1. Be cautious not to accidentally delete these configuration files.

This is important information because our database could be rendered inoperable if files are deleted. Now that it's installed, it's time to proceed with creating a database. But there is one step we have to do first—we need to configure the **listener**.

Configuring the listener

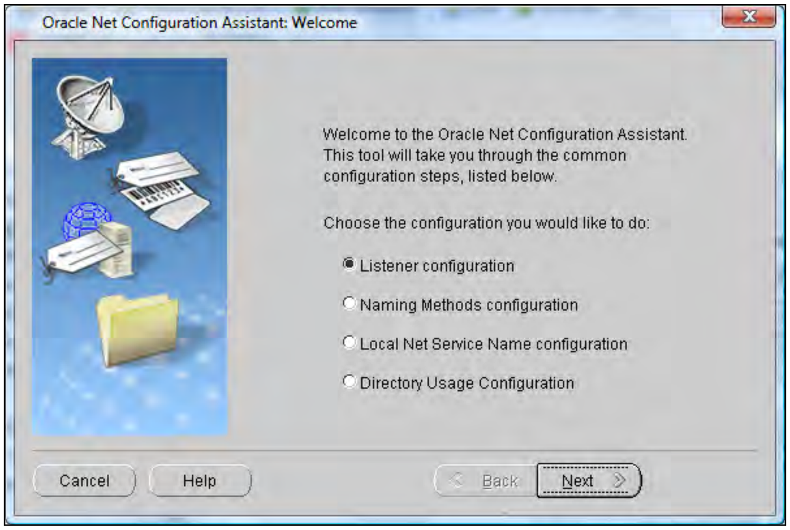
The listener is the utility that runs constantly in the background on the database server, listening for client connection requests to the database and handling them. It can be installed either before or after the creation of a database, but there is one option during the database creation that requires the listener to be configured—so we'll configure it now, before we create the database.

Run **Net Configuration Assistant** to configure the listener. It is available under the **Oracle** menu on the Windows **Start** menu as shown in the following image:


Oracle - OraDb11g_home1
Application Development
Configuration and Migration Tools
Administration Assistant for Win
Database Configuration Assistan
Database Upgrade Assistant
Locale Builder
Microsoft ODBC Administrator
Net Configuration Assistant
Net Manager
Integrated Management Tools
Oracle Installation Products
Warehouse Builder

The welcome screen will offer us four tasks that we can perform with this assistant. We'll select the first one to configure the listener, as shown here:

Download at Boykma.Com



Oracle Net Configuration Assistant: Welcome

Welcome to the Oracle Net Configuration Assistant. This tool will take you through the common configuration steps, listed below.

Choose the configuration you would like to do:

- ● Listener configuration
- ○ Naming Methods configuration
- ○ Local Net Service Name configuration
- ○ Directory Usage Configuration

The next screen will ask you what we want to do with the listener. The four options are as follows:

- **Add**
- **Reconfigure**
- **Delete**
- **Rename**

Only the **Add** option will be available since we are installing Oracle for the first time. The remainder of the options will be grayed out and will be unavailable for selection. If they are not, then there is a listener already configured and we can proceed to the next section—*Creating the database*.

For those of us installing for the first time on our machines, we need to proceed with the configuration. The next screen will ask us what we want to name the listener. It will have **LISTENER** entered by default and that's a fine name, which states exactly what it is, so let's leave it at that and proceed.

The next screen is the protocol selection screen. It will have **TCP** already selected for us, which is what most installations will require. This is the standard communications protocol in use on the Internet and in most local networks. Leave that selected and proceed to the next screen to select the port number to use. The default port number is 1521, which is standard for communicating with Oracle databases and is the one most familiar to anyone who has ever worked with an Oracle database. So, change it only if you want to annoy the Oracle people in your organization who have all memorized the default Oracle port of 1521.

Download at Boykma.Com

To change or not change the default listener port

Putting aside the annoyance, the Oracle people might have to suffer as there are valid security reasons why we might want to change that port number. Since it is so common, the people accustomed to working with the Oracle database aren't the only people who know that port number. Hackers looking to break into an Oracle database are going to go straight for that port number, so if we change it to something obscure, the database will be harder to find on the network for the people with malicious intent. If it does get changed, be sure to make a note of the assigned number.

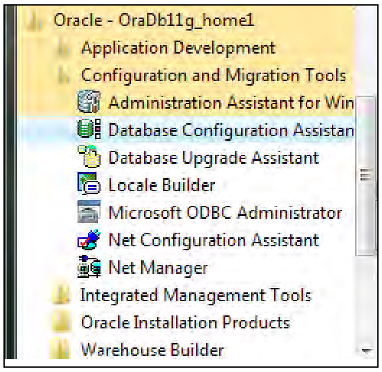
There also may be firewall issues that allow only certain port numbers to be open through the firewall, which means communication on any of the other port numbers would be blocked. 1521 might be allowed by default since it is common for the Oracle database. It would be a good idea to check with the network support personnel to get their recommendation.

That is the last step. It will ask us if we want to configure another listener. Since we only need one, we'll answer "no" and finish out the screens by clicking on the **Finish** button back on the main screen.

Creating the database

So far we have the Oracle software installed and a listener configured, but we have not created a database. We chose not to install the starter database because that defaults to a general purpose transactional database, and we want one that is oriented toward a data warehouse.

We can install a new database using **Database Configuration Assistant**, which Oracle provides to walk us step-by-step through the process of creating a database. It is launched from the Windows **Start** menu as shown in the following image:


Oracle - OraDb11g_home1
Application Development
Configuration and Migration Tools
Administration Assistant for Win
Database Configuration Assistan
Database Upgrade Assistant
Locale Builder
Microsoft ODBC Administrator
Net Configuration Assistant
Net Manager
Integrated Management Tools
Oracle Installation Products
Warehouse Builder

Download at Boykma.Com

Running this application may require patience as we have to wait for the application to load after it's selected. Depending on the system it is running on, it can take over a minute to display, during which there is no indication that anything is happening. It may be tempting to just select it again from the **Start** menu because it appears it didn't work the first time, but don't as that will just end up running two instances of the program. It will appear soon. The following are steps in the creation process:

1. The first step is to specify what action to take. Since we do not have a database created, we'll select the **Create a Database** option in Step 1. If there was a database already created, the options for configuring a database or deleting a database would be selectable. Templates can be managed with the **Database Configuration Assistant** application, which are files containing preset options for various database configurations. Pre-supplied templates are provided with the application, and the application has the ability to custom-build templates.

   Automatic Storage Management can be configured as well. It is Oracle's feature for databases for automatically managing the layout and storage of database files on the system. These are both topics for a more advance book on the Oracle Database. We will be creating a database using an existing template.

2. This step will offer the following three options for a database template to select:
   - **General Purpose or Transaction Processing**
   - **Custom Database**
   - **Data Warehouse**

   We are going to choose the **Data Warehouse** option for our purposes. If we already had a database installed that we wanted to use for learning OWB, but that's not configured as a data warehouse, it's not a problem. We can still run OWB hosted on it and create the data warehouse schema (database user and tables), which we'll be creating as we proceed through the book. This would be fine for learning purposes, but for production-ready data warehouses a database configured specifically as a data warehouse should be used.

3. This step of the database creation will ask for a database name. The name of the database must be one to eight characters in length. Any more than that will generate an error when trying to proceed to the next screen. This is an Oracle database limitation. The database name can also include the network domain name of the domain of the host it is running on, to further uniquely identify it. Follow the name with a period and then the domain, which itself can include additional periods.

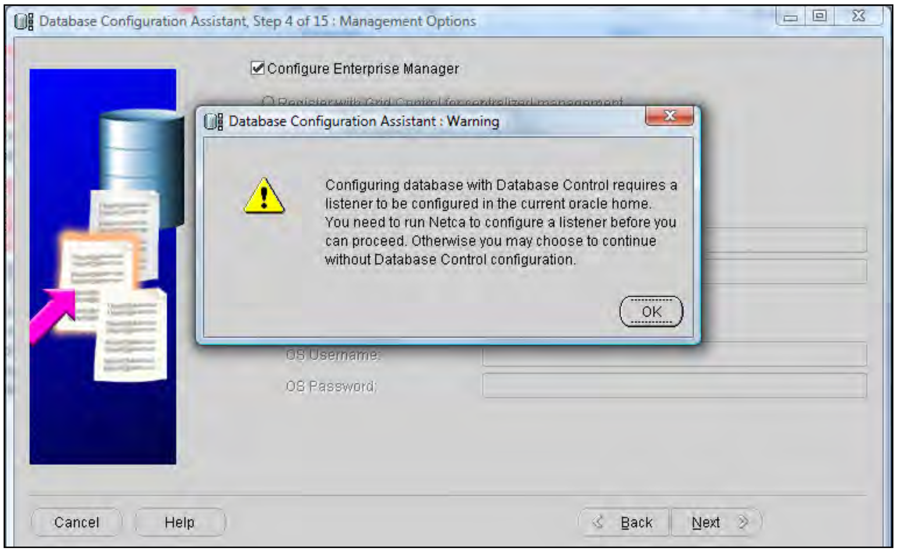
Download at Boykma.Com

If this database is being created for business use, a good naming scheme would reflect the purpose of the database. Since we're creating this database for the data warehouse of ACME Toys and Gizmos Company, we'll choose a name that reflects this—ACME for the company name and DW for data warehouse, resulting in a database name of `ACMEDW`. It is important to remember this name as it will be a part of any future connections to the database.

As the database name is typed in, the **SID** (or Oracle **System Identifier**) is automatically filled in to match it. If the domain is added to the database name, the SID will stop pre-populating after the first period is entered. The end result is that the SID becomes the same as the first part of the database name.

4. This step of the database creation process asks whether we want to configure **Enterprise Manager**. The box is checked by default and leave it as is. This is a web-based utility Oracle provides for controlling a database, and as it is very useful to have, we will want to enable it. There are two options for controlling a database: registering with **Grid Control** or local management. Grid Control is Oracle's centralized feature for controlling a grid, a network of loosely coupled modular hardware and software components that can be joined and rejoined together on demand to meet business needs. That is what the "*g*" in Oracle Database 11*g* stands for. If your network is not configured in a grid architecture, or you are installing on a standalone machine, then choose the local management option. It will automatically detect a Grid Control agent that is running locally, and if it doesn't find one, the **Grid Control** option will be grayed out anyway. In that case, you will only be able to select local management.

   When the **Next** button is clicked, the following message may appear:


Database Configuration Assistant, Step 4 of 15 : Management Options
Configure Enterprise Manager
Register with Grid Control for centralized management
Database Configuration Assistant : Warning
OK
OS Username:
OS Password:
Cancel
Help
Back
Next

Configuring database with Database Control requires a listener to be configured in the current oracle home. You need to run Netca to configure a listener before you can proceed. Otherwise you may choose to continue without Database Control configuration.

Download at Boykma.Com

That means a listener was not configured before creating the database. If this happens, we'll have to just pause our database creation and go back to the previous section about installing the listener and then come right back to this spot. There is no need to exit out of the database install window while doing this; just leave it on step 4. When we've completed the listener configuration, this screen will allow us to proceed to the next screen without that warning popping up again.

5. On this screen (step 5) we can set the database passwords on the system accounts using a different one for each account, or by choosing one password for all four. We're going to set a single password on all four, but for added security in a production environment, it is a good idea to make a different password for each. Click on the option to **Use the Same Administrative Password for All Accounts** and enter a password. This is very important to remember as these are key system accounts used for database administrative control.
6. This step is about storage. We'll leave it at the default of **File System** for storage management. The other two options are for more advanced installations that have greater storage needs.
7. This step is for specifying the locations where database files are to be created. This can be left at the default for simplicity (which uses the locations specified in the template and follows the OFA standard for naming folders described above). A storage screen will come up where we'll be able to change the actual file locations if we want, for all but the **Oracle-Managed Files** option.

The Oracle-Managed Files option is provided by the database so that we can let Oracle automatically name and locate our data files. A folder location is specified on the step 7 screen, which will become the default location for any files created using this option. This is why we won't be able to change any file locations later on during the installation if this option is chosen. However, files can still be created with explicit names and locations after the database is running.

8. The next screen is for configuring recovery options. We're up to step 8 now. If we were installing a production database, we would want to make sure to use the **Flash Recovery** option and to **Enable Archiving**. Flash Recovery is a feature Oracle has implemented in its database to provide a location that is managed by the database. It stores backups and files needed to recover a database in the event of disk failure. With **Flash Recovery Area** specified, we can recover data that would otherwise be lost in a system failure.

Download at Boykma.Com

Enabling archiving turns on the archive log mode of the database, which causes it to archive the **redo logs** (files containing information that is used by the database to recover transactions in the event of a failure.) Having redo logs archived means you can recover your database up to the time of the failure, and not just up to the time of the last backup.

These recovery options will consume more disk space, but will provide a recovery option in the event of a failure. Each individual will have to make the call for their particular situation whether that is needed or not.

We'll specify **Flash Recovery** and for simplicity, we will just leave the default for size and location. We will not enable archiving at this point. These options can always be modified after the database is running, so this is not the last chance to set them.

9. This step is where we can have the installation program create some sample schemas in the database for our reference, and specify any custom scripts to run. The text on the screen can be read to decide whether they are needed or not. We don't need either of these for this book, so it doesn't matter which option we choose.
10. The next screen is for **Initialization Parameters**. These are the settings that are put in place to define various options for the database such as **Memory** options. There are over 200 different parameters and to go through all of them would take much more time and space than we have here. There is no need for that at this point as there are about 28 parameters that Oracle says are basic parameters that every database installation should set. We're just going to leave the defaults set on this screen, which will set the basic parameters for us based on the amount of memory and disk space detected on our machine. We'll just move on from here. Once again, these can all be adjusted later after the database is created and running if we need to make changes.
11. The next screen is for security settings. For the purposes of this book and its examples, we'll check the box to **Revert to pre-11g security settings** since we don't need the additional features. However, for a production environment, it is a good idea to leave the default checked to use Oracle's more advanced security features.
12. This step is automatic maintenance and we'll deselect that option and move on, since we don't need that additional functionality. **Automatic Maintenance Tasks** are tasks that run in predefined maintenance windows of time to perform various preconfigured maintenance operations on the database. Since the database for this book is only for learning purposes, it is not critical that these maintenance tasks be done automatically.

Download at Boykma.Com

Automatic maintenance is designed to run during preset maintenance windows, which are usually in the middle of the night. So if the database system is shut down every day, there wouldn't be a good window to run the tasks on regularly anyway. If installing in a production environment with servers that will be running 24 hours a day every day, then consider setting up the automatic maintenance to occur. Oracle provides three pre-configured maintenance tasks to choose from—collecting statistics for the query optimizer (for improving performance of SQL queries), **Automatic Segment Advisor** for analyzing storage space for areas that can possibly be reclaimed for use, and the **Automatic SQL Tuning Advisor** for automatically analyzing SQL statements for performance improvements.

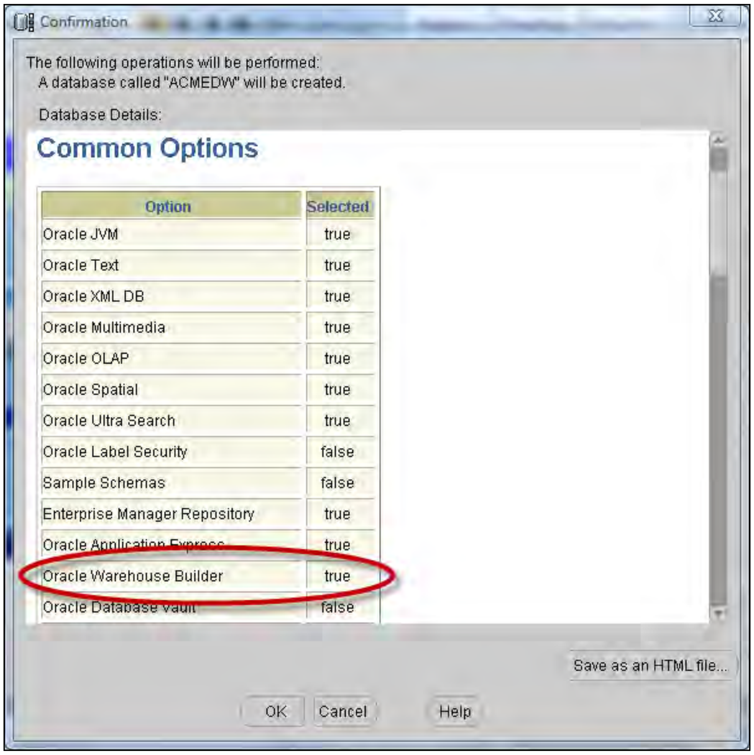
13. The next step (step 13 of 14) is the **Database Storage** screen referred to earlier. Here the locations of the pre-built data and control files can be changed if needed. They should be left set to the default for simplicity since this won't be a production database. For a production environment, we would want to consider storing datafiles on separate partitions for performance reasons, and to minimize the impact of disk failures on the running database if something goes wrong. If all the datafiles are on one drive and it goes bad, then the whole database is down.
14. The final step has the following three options, and any or all can be selected for creating the database:
    - Create the database directly
    - Save the creation options as a template for later use
    - Save database creation scripts that can be used later to create the database

    We'll leave the first checkbox checked to go ahead and create the database.

The **Next** button is grayed out since this is the last screen. So click on the **Finish** button to begin creating the database using the selections we've just chosen. It will display a summary screen showing what options it will be using to install with. We can save this as an HTML file if we'd like to keep a record of it for future reference.

Download at Boykma.Com

All that information will be available in the database by querying system tables later, but it's nice to have it all summarized in one file. We can scroll down that window and verify the various options that will be installed, including **Oracle Warehouse Builder**, which will have a **true** in the **Selected** column as shown here:



The following operations will be performed:
A database called "ACMEDW" will be created.

Database Details:

Common Options

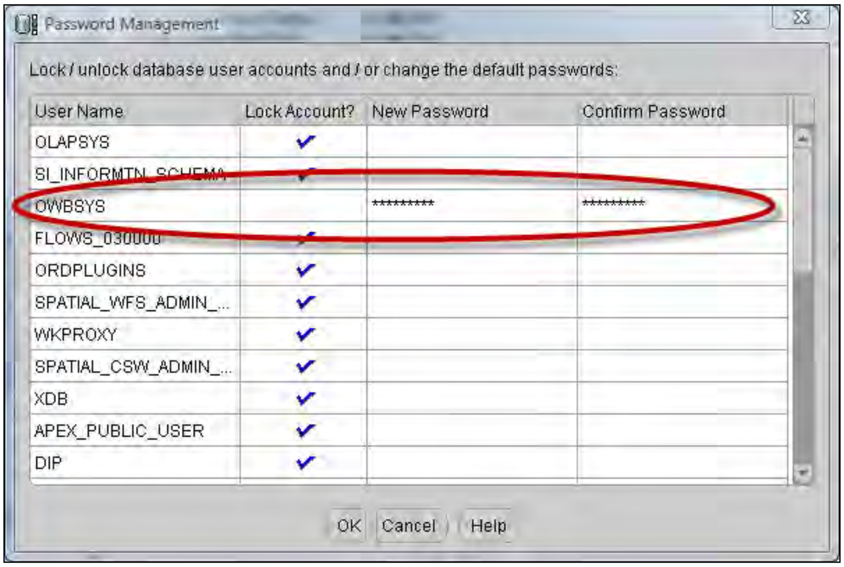
Option	Selected
Oracle JVM	true
Oracle Text	true
Oracle XML DB	true
Oracle Multimedia	true
Oracle OLAP	true
Oracle Spatial	true
Oracle Ultra Search	true
Oracle Label Security	false
Sample Schemas	false
Enterprise Manager Repository	true
Oracle Application Express	true
Oracle Warehouse Builder	true
Oracle Database Vault	false

We will be presented with the progress screen next that will show us the progress as it creates the database.

When the install progress screen gets to 100% and all the items are checked off, we will be presented with a screen summarizing the database configuration details. Take a screen capture of this screen or write down the details because it's good to know information on how the database is configured. Especially, we'll need the database name in later installation steps. We may see the progress screen at 100% doing nothing with apparently no other display visible. Just look around the desktop underneath other windows for the **Database Configuration Screen**. It's important for the next step.

Download at Boykma.Com

On the final Database Configuration Screen, there is a button in the lower right corner labeled **Password Management**. We need to click on this button to unlock the schema created for OWB use. Oracle configures its databases with most of the pre-installed schemas locked, and so users cannot access them. It is necessary to unlock them specifically, and assign our own passwords to them if we need to use them. One of them is the **OWBSYS** schema. This is the schema that the installation program automatically installs to support the Warehouse Builder. We will be making use of it later when we start running OWB. Click on the **Password Management** button and on the resulting Password Management screen, we'll scroll down until we see the **OWBSYS schema** and click on the check box to uncheck it (indicating we want it unlocked) and then type in a password and confirm it as shown in the following image:



Lock / unlock database user accounts and / or change the default passwords:

User Name	Lock Account?	New Password	Confirm Password
OLAPSYS	✓		
SI_INFORMTN_SCHEMA	✓		
OWBSYS		**********	**********
FLOWS_030000	✓		
ORDPLUGINS	✓		
SPATIAL_WFS_ADMIN_...	✓		
WKPROXY	✓		
SPATIAL_CSW_ADMIN_...	✓		
XDB	✓		
APEX_PUBLIC_USER	✓		
DIP	✓		

Click on the **OK** button to apply these changes and close the window. On the Database Configuration Screen, click on the **Exit** button to exit out of the Database Configuration Assistant.

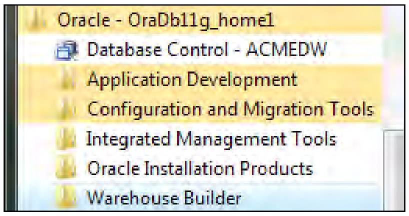
That's it. We're done installing our first database and it's ready to use. Next, we'll discuss installing the OWB client if we want to run the client on another computer, or if we already have a 10*g*R2 database installed that we want to use with the Warehouse Builder.

Download at Boykma.Com

There is a known bug that can occur when running Oracle Database Control version 11.1.0.6. This is the version that we're installing for this book. It occurs only on Windows Vista 32-bit and affects only the Database Control application, not the database itself. While the Database Control service is running, you will occasionally see a popup with the message **nmefwmi.exe has stopped working**. This is apparently a rather harmless error. To keep it from occurring, you can stop the Enterprise Manager Database Control service. It is accessible from the **Administrative Tools | Services** menu entry, and is the service named **OracleDBConsoleACMEDW**, where the **ACMEDW** is the SID for the database. If you need to use the Database Control application, you can start it up, do what you need to do, and then stop it. Oracle has reported that this bug is fixed in version 11.1.0.7.

Installing the OWB standalone software

If we are going to run the OWB client on the same computer as we just installed the Oracle database on, we don't need any more installations. That is the configuration used in this book. The OWB client software is now installed by default with the main database installation. We can verify that by checking the **Start** menu entry for Oracle. We will see a submenu entry for **Warehouse Builder** as shown in the following image:


Oracle - OraDb11g_home1
Database Control - ACMEDW
Application Development
Configuration and Migration Tools
Integrated Management Tools
Oracle Installation Products
Warehouse Builder

If we want to run the OWB client on another computer on the network, or if we have an older version of the database already installed (10*g* Rel 2) and want to be able to use the Warehouse Builder software with it, we'll need to continue here with the installation of the OWB client software. For all others, we can proceed to the next section on OWB—*OWB components and architecture*.

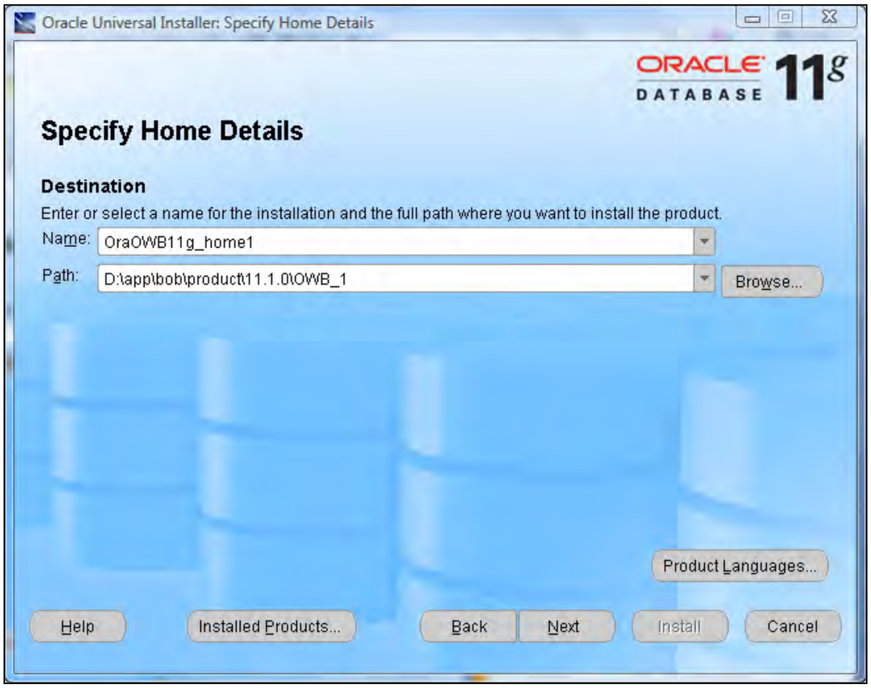
For the task of installing the standalone client, we'll need to download the OWB client install file. So we will go back to the Oracle site on the Internet. The download page is at the following URL at the time of writing: `http://www.oracle.com/technology/software/products/warehouse/index.html`. If that is not working, go to the main Oracle site and search for the **Business Intelligence | Data Warehousing** page where there is a link for the download of the OWB client.

Download at Boykma.Com

Once again we'll have to accept the license agreement before the download links will become active. So we'll accept it and download the install file to the client computer on which we'll be installing the software. The Windows ZIP file is about 1.1 GB in size so we need to make sure we have enough room on our hard drive to store the file. We'll need at least double that amount of space because the install files will take up that much space when unzipped.

When we have downloaded the ZIP file and unzipped it to our hard drive, run `setup.exe` in the top-level folder to run the Oracle Universal Installer. It should look familiar. Oracle is definitely correct in calling their installer "Universal". Every Oracle database product uses that installer, so we will become very familiar with it if we have to install any more Oracle products. It is universal also in the fact that it runs on every platform that Oracle supports, and so the same interface is used no matter where we install it. The installation steps are as follows:

1. The first step it goes through is asking us for the Oracle home details. It's similar to what it asked at the beginning of the database installation as shown in the following image:



Oracle Universal Installer: Specify Home Details

Specify Home Details

Destination

Enter or select a name for the installation and the full path where you want to install the product.

Name: OraOWB11g_home1

Path: D:\app\bob\product\11.1.0\OWB_1

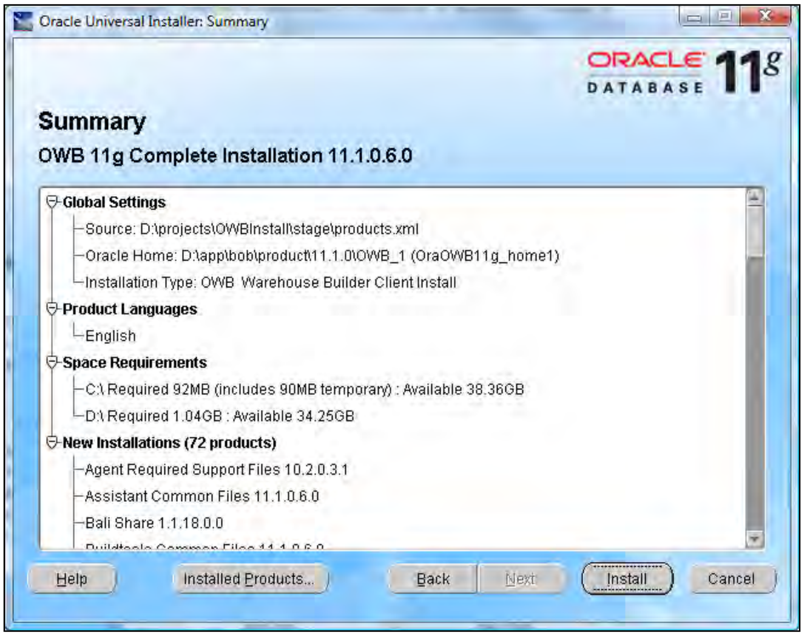
Download at Boykma.Com

The installer will again suggest **OraDb11g_home1** or something similar, but we'll change it to **OraOWB11g_home1** since it's just the OWB installation and not the full database.

> When installing the standalone OWB software, remember that it cannot be installed into the same ORACLE home as the database. It must reside in it own Oracle home folder. So if we have a database that's already installed on the same machine, we'll have to make sure the ORACLE_HOME we specify is different. The installer will warn us if we try to specify the same one and won't let us continue until it is different.

We need to verify the installation location for the home location also. The suggested name that it provides conforms to the OFA standard just as the database installation did, so we'll want to just change the drive letter if needed. However, the bottommost folder name can be changed if needed without violating the OFA standard. If it has a default of **db_1**, we can change it to **OWB_1** just to be clear that it's the OWB client.

2. The next and final step is the summary screen. The OWB client installation is not as complex as a full database installation, so it does not need all the additional information it asked for during the database installation. The summary screen should look similar to the following:



Summary

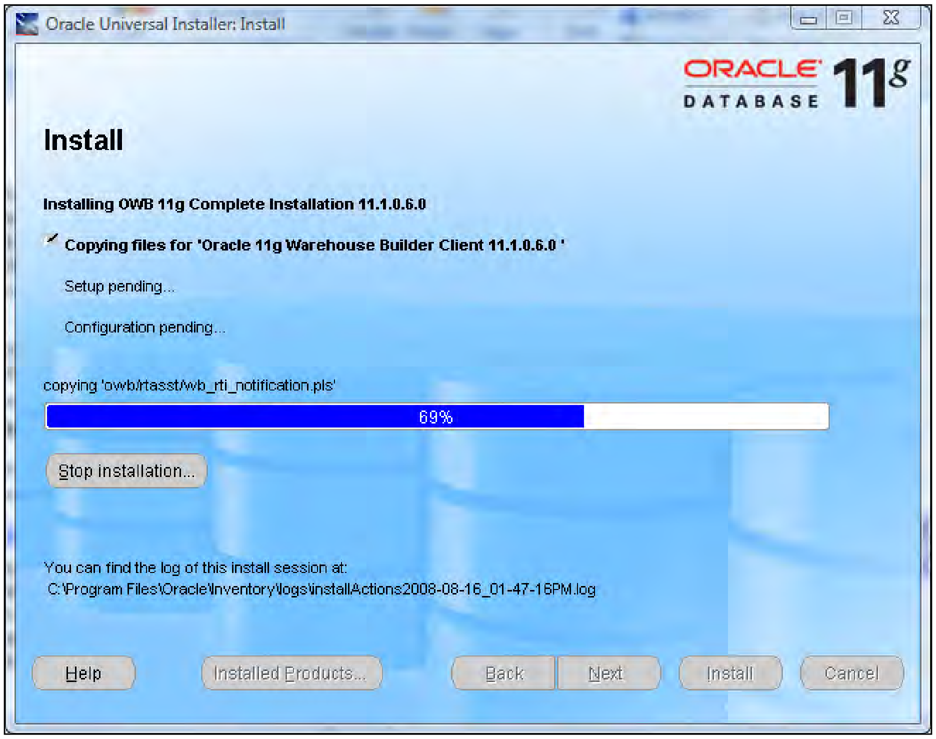
OWB 11g Complete Installation 11.1.0.6.0

```
Global Settings
  Source: D:\projects\OWBInstall\stage\products.xml
  Oracle Home: D:\app\bob\product\11.1.0\OWB_1 (OraOWB11g_home1)
  Installation Type: OWB Warehouse Builder Client Install
Product Languages
  English
Space Requirements
  C:\ Required 92MB (includes 90MB temporary) : Available 38.36GB
  D:\ Required 1.04GB : Available 34.25GB
New Installations (72 products)
  Agent Required Support Files 10.2.0.3.1
  Assistant Common Files 11.1.0.6.0
  Bali Share 1.1.18.0.0
```

Download at Boykma.Com

This summary gives us an idea of the disk space it will need, as well as the products that will be installed. If we scroll down the list, we'll see a number of other Oracle utilities and applications that it will install. We will also see items that are installed on the server as a part of the database install, but that will now be available to us on our client workstation. **SQL*Plus** appears there, which is the command line utility for accessing an Oracle database directly using **SQL** (**Structured Query Language**, the language used for accessing information stored in databases) among a host of other features.

Upon proceeding, the next screen will begin the installation and present us with the progress screen with a sliding bar moving to the right to indicate how far it has progressed. This is similar to what it did for the full database installation. An example of that screen is included next for reference:



Installing OWB 11g Complete Installation 11.1.0.6.0

Copying files for 'Oracle 11g Warehouse Builder Client 11.1.0.6.0 '

Setup pending...

Configuration pending...

copying 'owb/rtasst/wb_rti_notification.pls'

69%

You can find the log of this install session at:
C:\Program Files\Oracle\Inventory\logs\installActions2008-08-16_01-47-16PM.log

Install results

The log files with the results of the installation are stored in the same location as they are for a full database install. The universal installer will use that same folder for all its installs.

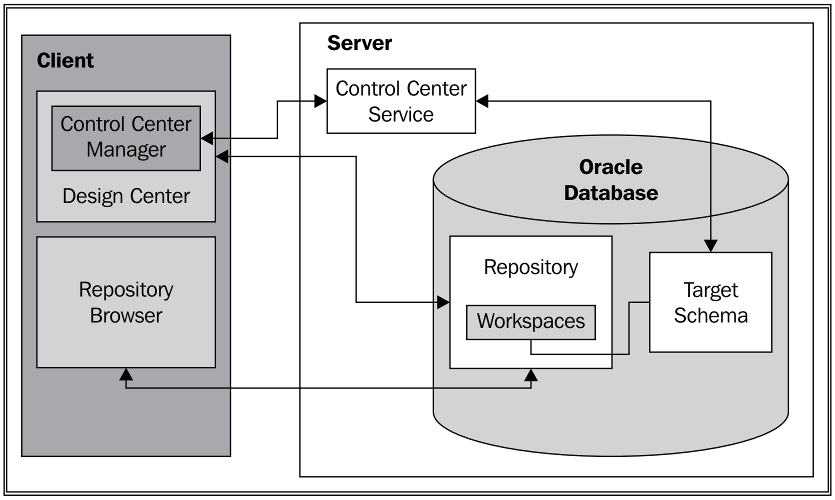
Download at Boykma.Com

When the installation is complete, we will be presented with the final success screen and an **Exit** button. And as if to remind us about the universal nature of the installer, it will pop up a confirmation box asking if we really want to exit, even though for this installation there is nothing else that would be available to do on that final screen if we said no.

OWB components and architecture

Now that we've installed the database and OWB client, let's talk about the various components that have just been installed that are a part of the OWB and the architecture of OWB in the database. Then we'll perform one final task that is required before using OWB to create our data warehouse.

Oracle Warehouse Builder is composed on the client of the **Design Center** (including the **Control Center Manager**) and the **Repository Browser**. The server components are the **Control Center Service**, the **Repository** (including **Workspaces**), and the **Target Schema**. A diagram illustrating the various components and their interactions follows:



Client and server

The previous diagram depicts a client and server, but these are really just logical notions to indicate the purpose of the individual components and are not necessarily physically separate machines. The client components are installed with the database as we've seen previously and, therefore, can run on the same machine as the database. This configuration is assumed throughout the book.

Download at Boykma.Com

The Design Center is the main client graphical interface for designing our data warehouse. This is where we will spend a good deal of time to define our sources and targets, and describe the **extract**, **transform**, **and load** (**ETL**) processes we use to load the target from the sources. The ETL procedures are what we will define to carry out the extraction of the data from our sources, any transformations needed on it and subsequent loading into the data warehouse. What we will create in the Design Center is a logical design only, not a physical implementation. This logical design will be stored behind the scenes in a Workspace in the Repository on the server. The user interacts with the Design Center, which stores all its work in a Repository Workspace.

We will use the Control Center Manager for managing the creation of that physical implementation by deploying the designs we've created into the Target Schema. The process of deployment is OWB's method for creating physical objects from the logical definitions created using the Design Center. We'll then use the Control Center Manager to execute the design by running the code associated with the ETL that we've designed. The Control Center Manager interacts behind the scenes with the Control Center Service, which runs on the server as shown in the previous image. The user directly interacts with the Control Center Manager and the Design Center only.

The Target Schema is where OWB will deploy the objects to, and where the execution of the ETL processes that load our data warehouse will take place. It is the actual data warehouse schema that gets built. It contains the objects that were designed in the Design Center, as well as the ETL code to load those objects. The Target Schema is not an actual Warehouse Builder software component, but is a part of the Oracle Database. However, it will contain Warehouse Builder components such as synonyms that will allow the ETL mappings to access objects in the Repository.

The Repository is the schema that hosts the design **metadata** definitions we create for our sources, targets, and ETL processes. Metadata is basically data about data. We will be defining sources, targets, and ETL processes using the Design Center and the information about what we have defined (the metadata) is stored in the Repository.

Download at Boykma.Com

The Repository is a Warehouse Builder software component for which a separate schema is created when the database is installed—OWBSYS. This is the schema we talked about unlocking during the installation discussion previously as one of the final steps in the database creation process. This will be created automatically by the 11*g* install, but is installed separately using scripts if we want to host the Repository on an Oracle 10*g* database. The explanations in this book all assume that the Repository is hosted on an Oracle 11*g* database. The *Oracle Warehouse Builder Installation and Administration Guide* found at the following URL: `http://download.oracle.com/docs/cd/B28359_01/owb.111/b31280/toc.htm` discusses the procedure for installing the Repository schema on an Oracle 10*g* release 2 database if needed.

The Repository will contain one or more Workspaces as shown in the previous diagram. A Workspace is where we will do our work to create the data warehouse. There can be more than one workspace defined in the Repository. A common example of how multiple workspaces can be employed is to use different workspaces corresponding to sets of users working on related projects. We could have one workspace for development, one for testing, and one for production. The development team could be working in the development environment separately from the test team that would be working in the test environment. For our purposes at the ACME Toys and Gizmos Company, we will be working out of one workspace.

This concept of the workspace is new in this latest release of OWB. The Repository is created in the OWBSYS schema during the database installation. So setting up the Repository information and workspaces no longer requires **SYSDBA** privileges for the user to install the Repository. SYSDBA is an advanced administrative privilege that is assigned to a user in an Oracle database. This allows the user to perform tasks affecting the database and other database users that ordinary user accounts cannot do (or for that matter, other administrative accounts without SYSDBA). For security reasons, we want to restrict user accounts with SYSDBA privilege to a minimum. So it is good that we don't have to use that privilege when we install the Repository.

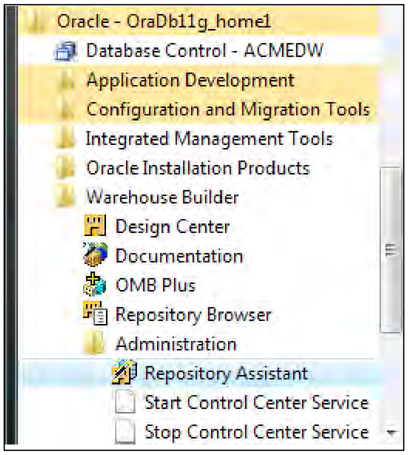
One final OWB component to consider is the Repository Browser on the client. It is a web browser interface for retrieving information from the Repository. It will allow us to view the metadata, create reports, and audit runtime operations. It is the only other component besides the Design Center and the Control Center Manager that the user interacts with directly.

We will have a chance to visit each one of these areas in much more detail as we progress through the design and build of our data warehouse. However, first there is one more installation step we have to take before we can begin using the Warehouse Builder. The Repository must be configured for use and a workspace must be defined.

Download at Boykma.Com

Configuring the repository and workspaces

We have talked about the OWBSYS schema that is created for us automatically during the Oracle 11*g* installation, and we have also looked at unlocking it and assigning a password to it. However, if we were to connect to the database right now as that user, we would find that as yet no objects exist in that schema. That is what will be done during this final installation step. We are going to use the **Repository Assistant** application to configure the repository, create a workspace, and create the objects in the repository that are needed for OWB to run. This application is available from the **Start** Menu under the **Warehouse Builder | Administration** submenu of the Oracle program group as shown here:


Oracle - OraDb11g_home1
Database Control - ACMEDW
Application Development
Configuration and Migration Tools
Integrated Management Tools
Oracle Installation Products
Warehouse Builder
Design Center
Documentation
OMB Plus
Repository Browser
Administration
Repository Assistant
Start Control Center Service
Stop Control Center Service

These menu options will appear locally on a client if we've installed the standalone Warehouse Builder client, as well as on the server. So where should we run the **Repository Assistant** if we have both? The most common configuration is to run this application on the same machine where the repository is located and the Control Center Service is going to run, which is all on one machine. There are other less common options for where to run the Control Center Service and where the Repository is located in relation to the target schema. These options are documented in *Oracle Warehouse Builder Installation and Administration Guide*. The URL for the chapter in the guide is the following: `http://download.oracle.com/docs/cd/B28359_01/owb.111/b31280/install_rep02.htm#BABJDHCJ`. This information can also be found in Chapter 2, in the subsection on implementing a remote runtime.

We want the runtime implemented on the server, which is the most common and simplest configuration. The Repository Assistant pops up an extra screen if it is running remotely from the client, which we will see next. We would see it during the installation if we were on a remote computer.

Download at Boykma.Com

The steps for configuration are as follows:

1. We'll launch the **Repository Assistant** application on the server (the only machine we've installed it on) and the first step it is going to ask us for the database connection information—**Host Name**, **Port Number**, and **Oracle Service Name**—or a **Net Service Name** for a **SQL*Net connection**. SQL*Net is Oracle's networking capability for communicating with databases in a distributed networked environment. A naming method is configured so that when using a Net Service name, SQL*Net will know what connection information to use for the connection. We have not configured a naming method, since we don't really need it just to connect locally, so we'll use the **Host Name**, **Port Number**, and **Oracle Service** name option as follows:
    - The **Host Name** is the name assigned to the computer on which we've installed the database, and we can just leave it at **LOCALHOST** since we're running it on the computer that has the database installed.
    - The **Port Number** is the one we assigned to the listener back when we had installed it. It defaults to the standard 1521. This is an example of why the issue of changing or not changing that default port number was mentioned. If we changed it but can't remember what we changed it to, then the following tip will help out.

Determining what port your listener is listening on

There are a couple of options we have for this. One is to perform the following steps:

Open a command prompt window and type in the following command:

```
C:\>lsnrctl
```

This will launch the **Listener Control** program, which is the command line utility Oracle provides for controlling the listener. Then enter the following command at the listener control prompt:

```
LSNRCTL> status
```

Look for the line that says:

```
Listening Endpoints Summary...
```

The next line will have the port number listed along with the protocol and host name such as the following:

```
(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=computer)
(PORT=1521)))
```

We can find information about the second option for determining the port number in the listener configuration file, `listener.ora`, in the Oracle home `NETWORK\ADMIN` directory. Open that file with Notepad and look for the above line.

Download at Boykma.Com

- For the **Service Name**, we will enter the name we assigned to our database during step 3 of the database creation process. The name we used is `ACMEDW`. At the end of the database configuration assistant process, a detail screen was displayed. It was suggested that it would be a good thing to take a screen capture of it because it contained details about the database configuration, which would be useful later. One of the items on that screen was the database name that was assigned. If that is not available, then here's another tip to find the database name.

Finding your database instance name

There are a number of places where the database name appears on the database server without us having to log in to the database. One is in the listener control program. Open a command prompt window and type in the following command:

```
C:\>lsnrctl
```

This will launch the Listener Control program. Then enter the following command at the listener control prompt:

```
LSNRCTL> service
```

Look for the instance name in the list of services that appears.

Another option is to check the name of the Windows service that is started for the database. The database service name is a part of that name. Open **Control Panel | Administrative Tools | Services**. The Windows service names for the Oracle processes all start with `Oracle`. The service that runs the actual database is named `OracleService<dbname>` where `<dbname>` is the name of the database instance that you are looking for. The name says `OracleServiceACMEDW` for a database name of `ACMEDW`.

We can also check the Oracle base folder, which is the folder where the Oracle software was installed. The `Admin` folder contains a folder named for the database instance if we followed the default naming conventions for folder names during the installation. That is one reason to stick with the OFA standard when installing Oracle products.

2. Now that we've determined the connection information for our database, we'll move along to step 2 of **Repository Assistant**. It asks us what option we'd like to perform of the following:
   - **Manage Warehouse Builder workspaces**
   - **Manage Warehouse Builder workspace users**
   - **Add display languages to repository**
   - **Register a Real Application Cluster instance**

Download at Boykma.Com

We're going to select the first option to manage workspaces and move along to the next step.

3. This step asks us what we'd like to do with workspaces: create a new workspace or drop an existing one. We'll select the first option to create a new workspace.
4. This brings us to step 4 of the process, which is to specify an owner for the workspace. We are presented with two options: to create a new user or to be the owner. To perform the first option, we will need to specify a database user who has DBA privileges that are required to be able to create a new user in the database. The second option is to specify an existing database user to become the owner of the workspace. This user must have the **OWB_USER** role assigned to be able to successfully designate it as a workspace owner. That is a database role required of any user who is to use the Warehouse Builder. If the existing user who is selected does not have that role, then it must be assigned to the user. An additional step will be required to specify another user who has the ability to do that assignment (grant that roll) or has DBA privileges. This second user must have the **Admin Option** specified for the OWB_USER role to be able to grant it if he or she does not have DBA privileges.

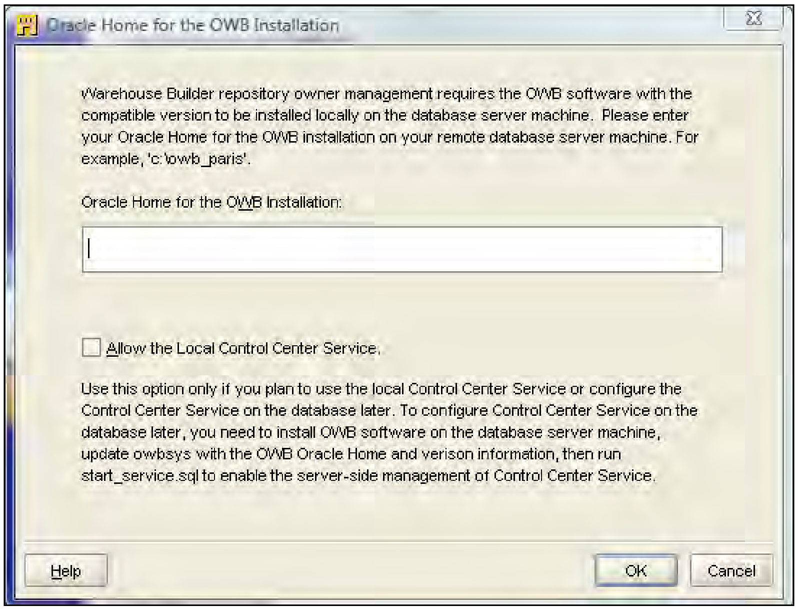
   The user specified here, whether new or existing, will become a deployment target for the Warehouse Builder. This means that the user will be able to access the Design Center for building the ETL processes and the Control Center Manager for deploying and auditing. We'll specify a new user for the ACME Toys and Gizmo's warehouse, since we've just installed this database and no other users are created yet.
5. This step will depend on which option we specified in step 4. If we are creating a new user, it will ask us for an existing user with DBA privileges in the database. The SYSTEM account is the default provided there, but if we have a different account that is a DBA in the database, we can use that. If we have specified an existing user in step 4, then step 5 will ask us for the username and password for that user, as well as the name of the new workspace to create.

   Since we're specifying a new user, we will put in the password for the system user and proceed to the next step. The password used here is the one we previously defined for the system accounts when we created our database.
6. In this step we specify the new username, password, and workspace name. We'll use **acmeowb** for the username and **acme_ws** for the workspace name.

Download at Boykma.Com

7. This step will ask for the password for the OWBSYS user. This schema was installed for OWB to use for the repository. The password it's looking for is the one we set up back on the final database configuration screen at the end of running **Database Configuration Assistant** to configure the database. This step will only be required upon first running **Repository Assistant** to create a new workspace since it also has to perform the process of initializing the repository in the OWBSYS schema first. That is a one-time process which is why subsequent runs of **Repository Assistant** to manage workspaces, will not require this step.

   After putting in that password, if we were running the **Repository Assistant** on a different machine than the database was installed on, then we would encounter the following screen. We referred to it earlier when talking about running the **Repository Assistant** remotely.



Oracle Home for the OWB Installation

Warehouse Builder repository owner management requires the OWB software with the compatible version to be installed locally on the database server machine. Please enter your Oracle Home for the OWB installation on your remote database server machine. For example, 'c:\owb_paris'.

Oracle Home for the OWB Installation:

Allow the Local Control Center Service.

Use this option only if you plan to use the local Control Center Service or configure the Control Center Service on the database later. To configure Control Center Service on the database later, you need to install OWB software on the database server machine, update owbsys with the OWB Oracle Home and verison information, then run start_service.sql to enable the server-side management of Control Center Service.

It doesn't know the location of the Oracle Home on the server, and so must prompt for it. It also provides the option for a **Local Control Center Service** that is for the remote runtime option discussed in the installation guide. Since we're running our database on the same machine as our client, we won't see this screen.

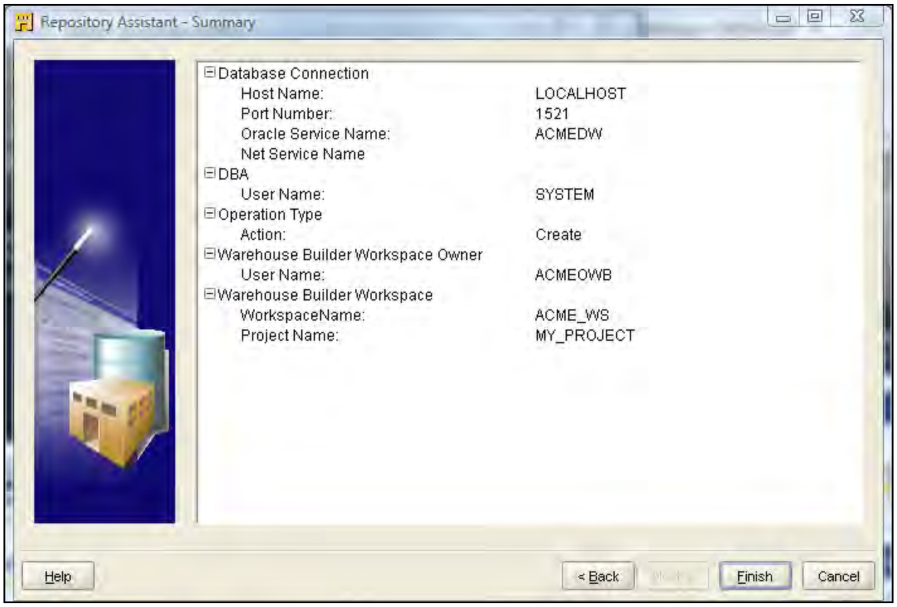
Download at Boykma.Com

8. This step asks for **tablespace** names for the OWBSYS schema. A tablespace is a logical entity in an Oracle database for storing data. All objects created are assigned to a tablespace, which stores the data physically in a datafile or datafiles assigned to the tablespace. The administration of tablespaces in an Oracle database is more than we have room for here, so we won't be creating any new tablespaces to hold the OWBSYS data. We'll just leave the defaults selected—the **USER** tablespace for data indexes and snapshots, and the **TEMP** tablespace for temporary data. For advanced production databases, it would be a good idea (at a minimum) to specify a separate tablespace for OWBSYS, and actually think about using three new tablespaces for those three that have the USER tablespace assigned.
9. This step is to select a base language for the repository, so we'll make the appropriate selection. Once the repository is created, we cannot change the base language and there can only be one base language assigned to the repository. Physical names of repository objects are assumed to be in the base language. The **Repository Assistant** will automatically assign the base language depending on the locale that is assigned to the computer we're installing on. We also have the option of selecting one or more display languages that will allow users to assign a business name to physical objects in their own language. Unlike the base language, we can assign display languages after the repository is created. Select any of those that apply.
10. We're almost finished. The final step is the optional step 10 to specify any workspace users from existing database users. We specified the workspace owner as a new user earlier in the install process, and now it's asking for any additional users who we might want to have access to the workspace. The workspace owner is allowed to add and remove database users from the workspace.

Removing a database user from the workspace does not delete that user account from Oracle. It only removes him or her as a valid user of the workspace.

After selecting any user, the **Repository Assistant** will present us with a summary screen of the actions it will take and the information we entered, as shown in the following image:

Download at Boykma.Com



⊟Database Connection	
Host Name:	LOCALHOST
Port Number:	1521
Oracle Service Name:	ACMEDW
Net Service Name	
⊟DBA	
User Name:	SYSTEM
⊟Operation Type	
Action:	Create
⊟Warehouse Builder Workspace Owner	
User Name:	ACMEOWB
⊟Warehouse Builder Workspace	
WorkspaceName:	ACME_WS
Project Name:	MY_PROJECT

Notice the name of the project at the end. There was no option to specify that project name, so it's just using a default name. It always sets up a default project in a new workspace by that name, but we can change it later when we actually start designing our data warehouse and working with the workspace in the Design Center.

Click on the **Finish** button and it will begin the installation, presenting us with a scroll bar moving to the right as it progresses through the installation. The very first time it runs, it will take around 5 to 10 minutes to run before reporting the success pop up, as it has to initialize the repository in the OWBSYS schema. Creating new workspaces after the first time will be very quick, taking no more than a few seconds to complete.

Download at Boykma.Com

Summary

That's it. We've gone through the install process of the Oracle 11*g* database. It automatically installs the Warehouse Builder components as well as the OWBSYS database user. We've also gone through a standalone installation of the OWB client on a separate workstation and have run **Repository Assistant** to configure our first workspace. We've also discussed the architecture of the Warehouse Builder components as they are now installed on our system. OWB is now installed and ready to use, so we can begin our project of designing and installing a data warehouse.

The general process we're going to follow throughout the rest of the book to actually build our data warehouse is to start by defining our data sources—where we will import the data from. We will import or define definitions of those sources, so that the Warehouse Builder knows about them. Then we will define our target data structures—where we will be loading data into during ETL and validate those structures. They will have to be generated and deployed to the target schema, which is the process of building the target. After that, comes the process of designing and implementing our ETL to load the target from the sources.

Now that we have the software and database loaded, it's time to begin by defining our sources of data.

Download at Boykma.Com

2
Defining and Importing Source Data Structures

The Warehouse Builder software and Oracle database have been installed, and we're ready to begin building our data warehouse. The first thing we have to do is define what our sources of data will be. If we are going to build a useful data warehouse, we have to know what kinds of information our users are going to need out of the warehouse. To know that, we have to know the following:

- The format in which the data is currently stored and where it is stored.
- Whether there is a **transactional database** currently in use or not, which supports day-to-day operations and from which we'll be pulling the data.

  A transactional database is different from a data warehouse database in that it is designed to support the day-to-day transactions that keep an organization running.
- Whether the database is an Oracle database or another vendor's database such as Microsoft SQL Server.
- Whether there are any **flat files** of information saved from database tables or other files that users keep, which might be a source of information

  A flat file is a file in text format that stores data in some kind of delimited format. The most common example of this kind of file is a **CSV** file, or a comma-separated file, that can be saved from a spreadsheet or extracted from a database table. It is called a flat file because it is in a text-only format and doesn't need to be interpreted by another program or application to read it.

The Warehouse Builder can help us with importing data from any (or all) of these formats into our data warehouse, and we're going to see how to do that in this chapter.

Download at Boykma.Com

Preliminary analysis

In any data warehouse project, we are going to need to do some up-front analysis to determine what data will need to be captured into our warehouse. The analysis will tell us where the data is located, and in what format, so that we can begin to define our source data structures in the **Warehouse Builder**. In our case, we will presume that we have interviewed the management at the ACME Toys and Gizmos company and they have indicated the following:

- The high-priority information that they would like to see from this data warehouse project is *sales-related data* for all their stores
- They don't have an idea about the comparative sales in the various stores, so they need some way to view all that data together to do an analysis that shows how well, or poorly, the stores are doing
- In the future, they would also like to be able to compare store sales with their web site sales, but that will not be required for this first data warehouse we build

We are doing a very simple analysis of our data warehouse project because the focus of the book is primarily on OWB. This book is all about using the Warehouse Builder and that begins *after* the initial analysis, and so we will cover just enough information to lay the groundwork for what follows. For more coverage of the design and analysis phase from a very practical standpoint, a very good book you should look up is *"The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling", Ralph Kimball and Margy Ross, John Wiley and Sons, Inc.* This book covers in detail the analysis and design considerations you should take into account when designing a data warehouse and uses practical examples from a number of industries.

ACME Toys and Gizmos source data

Talking to users, administrators, and database administrators in ACME has helped us discover that there is a transactional system in use (called a **Point-of-Sale** or **POS** system). This system supports the stores that ACME has located in various cities throughout the country, and in other countries in Europe and Asia.

Download at Boykma.Com

This system maintains data in a Microsoft SQL Server database named `ACME_POS`, and tracks individual sales transactions that occur for all of ACME's toys and gizmos. This database contains tables that store information about each sale along with all associated sales information such as the item sold, its price, the store in which it is sold, the register that processes the sale, and the employee who made the sale. Right away, we recognize that this would be a good source of data to help satisfy the management objective of analyzing their sales data better.

We have also found that the IT department that runs the web site for ACME Toys and Gizmos has its own database that supports the web site order management process. It is implemented in Oracle and handles the processing of all orders taken for products through the web site. It contains tables that store information about:

- The orders taken
- Information about the customer who placed the order
- Information about the individual products that were ordered

This is an example of the **source data** that might not be needed in the initial stages of a data warehouse project, but that could be requested by users at a later stage. Then we can expand the data warehouse implementation to include web site sales data. We will also take a look at importing source metadata for it later in this chapter.

Scripts have been provided on the Packt web site at `http://www.packtpub.com/files/code/5746_Code.zip`. These scripts can be used to build the Oracle web site orders database referred to in this chapter. We also have a CSV file in this code bundle that will be required to import metadata. We have a script to install the SQL Server database. Instructions to use these files can be found with the code bundle.

However, working on it beyond that is not in the scope of this book.

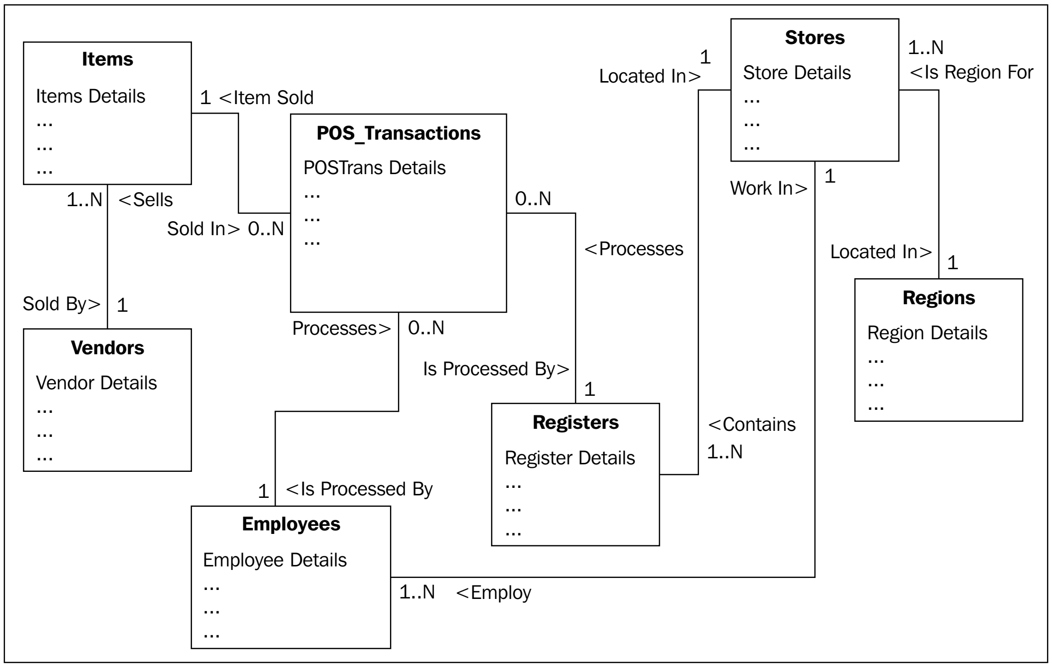
The POS transactional source database

The **DBA** (**Database Administrator**—the person responsible for the maintenance and administration of the database) is in charge of the POS transactional database. The DBA has provided an **Entity-Relationship** (**ER**) **diagram** of the database to help us understand the database and the relationships between the various tables. The diagram is in the **UML** (**Universal Modeling Language**) notation. The following image depicts a simplified version of the diagram containing the main tables of interest and the relationships between them, including the cardinalities. The **cardinality** indicates how the records in one table relate to records in the other. The cardinality can be expressed as many-to-many, one-to-many, many-to-one, or one-to-one, and is indicated in the diagram with counts composed of the following:

Download at Boykma.Com

- 0..N—zero or more
- 1..N—one or more
- 1—one only

The details about the columns in each table will be covered when we define the metadata for them. If you are familiar with ER diagrams, the process of implementing a database based on the diagram, and the concept of **normalization**, you can skip the following section and move on to the *The web site order management database* section.



The main table in the ACME_POS database is the `POS_Transactions` table. It holds information about each transaction that takes place in a store, including the cash register that processed the transaction, the employee who worked the register, the item sold, the quantity sold, and the date. However, not all of that information is stored directly in the `POS_Transactions` table; only the date and quantity are stored directly. If all the details about the item were included in every record in the `POS_Transactions` table, there would be a large amount of duplicated information. After all, the store is not going to sell an item only one time because if it did, it wouldn't be in business for long. There will be potentially hundreds to thousands of sales of the same item each day, depending on how busy a particular store is. With each of those sales, a row gets placed in the `POS_Transactions` table.

Download at Boykma.Com

We can see from the diagram that a separate table was created to hold item information and a link made from the main `POS_Transactions` table to the `Items` table. That link is created via a **foreign key** stored in the `POS_Transactions` table for the `Items` table. Instead of storing all the information about the item in the `POS_Transactions` table, a single column called the foreign key is placed there. This foreign key has a value corresponding to a value in the **primary key** column of the `Items` table. A primary key is a value that uniquely identifies a row in the table and, therefore, is not duplicated. We can then look up in the `Items` table for the information about the item for sale by using the value in the foreign key column for the item.

This concept of storing an item's information in a separate table results in a much greater accuracy of data, as we don't have to duplicate the item information. It is only entered *once* in the `Item` table. If it has to be updated, there is only one record in the `Item` table to update, and not thousands of records in the `POS_Transactions` table. This is known as database normalization. A transactional database is usually normalized due to this need for data accuracy.

The **attributes** of the `POS_Transactions` table are the individual pieces of information stored in it. Each of the attributes corresponds to one of the lines originating from the `POS_Transactions` table in the diagram, all except the `quantity` attribute. We can see that information about the employee who worked the register for the sale is stored in a separate table, the `Employees` table. This is similar to how the items sold are handled and stored in the `Items` table, as well as how the information about the register on which the sale was processed is stored in the `Registers` table.

In addition to these tables, we can also notice that a few other tables in the diagram are linked in various ways to these tables. They provide us with even more information about the attributes of a transaction and, therefore, about the transaction itself. We can see a table hanging off the `Items` table called `Vendors`. This table stores the information about each vendor who supplies toys and gizmos to the ACME company. A table called `Stores` is linked to the `Registers` table. This table tells us information about the store in which the register is located and, therefore, about the store that made the sale. Linked to the `Stores` table is the `Regions` table, which provides a location breakdown by region for the stores. ACME Toys and Gizmos is a worldwide operation and likes to track sales by breaking the world up into regions such as Europe and Asia, and for the US it's Northeast, Southwest, and so on.

Download at Boykma.Com

At this point we can begin to understand why the management found it so difficult to compare sales data from all their stores and web site, and why they would like to implement a data warehouse for their data explorations. Let's look at what kind of SQL query would be required to determine the number of flying discs sold today in Europe that were supplied by a particular vendor. Let's just consider the number of tables involved. We need the `POS_Transactions`, `Items`, `Vendors`, `Registers`, `Stores`, and `Regions` tables. The only table we don't need is the `Employees` table. But, why do we need the `Registers` and `Stores` tables when they wanted to know only the amount sold in the region? Well, the answer is that it's very tough for management to get the data they need. You see there is no direct connection from the `POS_Transactions` table to the `Regions` table.

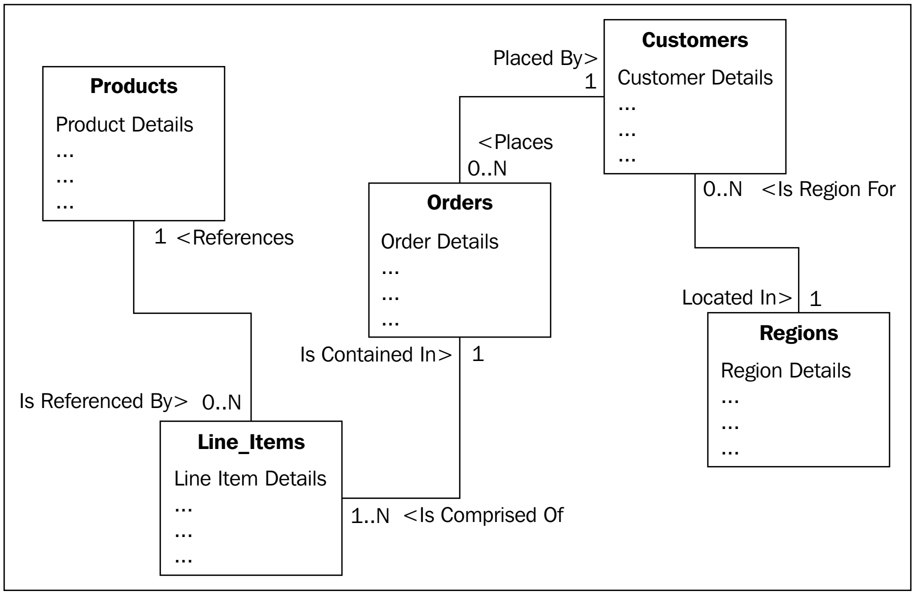
The only way to get the region for a given transaction is to look up the register that processed the transaction, and then look up the store in which the register is located and that store record will give you the region. All of this is done with one massive **join SQL query** to join all these tables together. A join query is one that pulls data from more than one table at a time. As the database has a normalized structure, we have to include those two additional tables in our join, which we really don't want, just to get to the information we want. If we're talking about millions of transactions, which is not at all an unreasonable situation for any large sales operation, we would end up with highly inefficient queries that take a long time to run and make management very unhappy with the database.

So, we're going to solve this problem with our data warehouse, which will have a much better organization of tables for querying as we'll see in the next chapter.

The web site order management database

The DBA in charge of the Oracle database for the web site order management system has provided us with its ER diagram for our information. As with the POS transaction database, an ER diagram is provided here in a shortened version to give us an idea of the tables involved and their relationships with each other. Later in this book, we will have examples dealing with the POS transactional database as it contains the sales data for the stores. This database is presented here because of the possible future requirements to include this data. We'll use it to provide an example of importing from a database, and as an example of some minor issues that can be encountered when trying to analyze multiple sources of data.

Download at Boykma.Com



This database holds the sales information for the web site, and we have to understand it before importing the database. We can see that it has some tables that are similar to the tables in the previous ER diagram that we just saw. The `Orders` table is the main table in this database instead of the `POS_Transactions` table. It holds information about customers who placed an order and a list of ordered items. The customer information includes the region in which the customer is located, and this is identical to the information in the `Regions` table in the POS Transactions database. The customer information is stored in a `Customers` table, which is linked to the `Orders` table, and we can see that the `Regions` table is linked to the `Orders` table through the `Customers` table. The list of ordered items is stored in the `Line_Items` table, which also has product information that identifies which product was ordered. The product information is stored in the `Products` table (which is similar to the `Items` table in the POS database). We can see that it has a link to the `Line_Items` table in our diagram.

Now we may get confused because this database has a `Line_Items` table and the other database has an `Items` table. But we're told that the `Products` table actually corresponds to the `Items` table, and not the `Line_Items` table. While this company is entirely fictional, this kind of issue of multiple departments, each with their own database and convention for naming tables and columns, is all too common in the real world for data warehouse projects. It's up to us to make sense out of it all and pull all that data into a single data warehouse where it can be queried at once. And this is what makes our job so interesting.

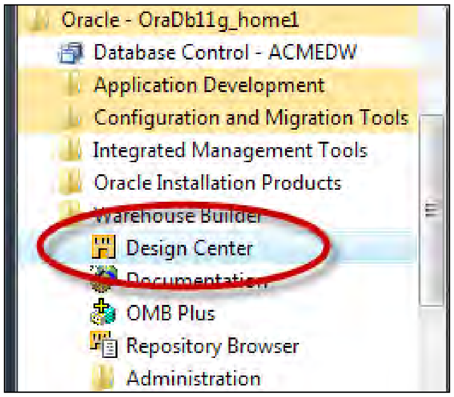
Download at Boykma.Com

Let's look at one more issue with this order management database before we move on. This issue is the relationship between the `Orders` table and the `Line_Items` table. Each order is composed of a variable number of line items of ordered products. One person may place an order on the web site for a doll and a fire truck, whereas another may order a game, a deck of cards, and a baseball bat. We've seen that this relationship between tables is accomplished in the database by storing a foreign key to the other table to indicate the relationship, but there could be any number of line items in an order. This would mean you will need any number of line item foreign keys stored in the `Orders` table, but that is not possible. The reason is that the foreign key in this situation is going the other way. The `Line_Items` table stores a foreign key to the order of which it is a part of as a line item can be associated with only one order.

This was a brief overview of the source data structures we're going to be working with and also a very brief introduction to some database design issues. Without further ado, let's turn our attention to the Warehouse Builder, which is the real subject of this book.

An overview of Warehouse Builder Design Center

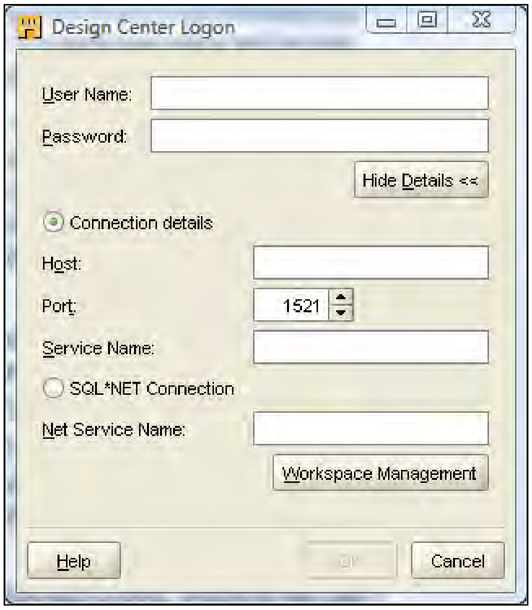
The **Design Center** is the main graphical interface that we will be using to design our data warehouse, but we also use it to define our data sources. So let's take some time at this point to go over the user interface and familiarize ourselves with it. We launch **Design Center** from the **Start** menu under the **Oracle** menu entry, as shown in the following image:



Download at Boykma.Com

The **Design Center** must connect to a workspace in our repository. To review briefly, we discussed the architecture of the Warehouse Builder in *Chapter 1*. This included the repository in which we created a workspace and a user, who would be the owner of the workspace. We used the Repository Assistant application to configure our repository and create that user. The repository is located in the OWBSYS schema that was the pre-installed schema the database installation provided for us. The user name chosen was **acmeowb** and the workspace name was **acme_ws**. Now it's time to make use of this user and workspace.

The first screen we'll be presented with is the **Logon** screen:



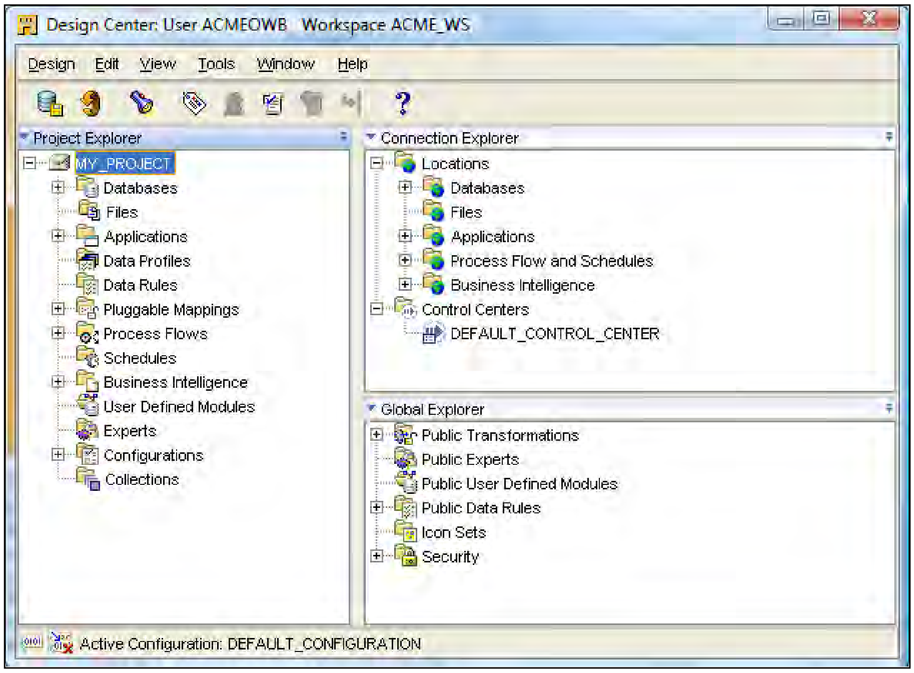
The first time we use this application, the **Logon** dialog box comes up all blank. But after we fill in our information for the first time, it will remember the **User Name** and **Connection details** on subsequent executions of the Design Center. Also, it will present us with a smaller version of the dialog box with just **User Name** and **Password**, so that we can just enter the password and don't have to re-enter the connection details. The button above the connection details that now displays **Hide Details <<** will display **Show Details >>**. If we need to change the connection details in that case or to just see what they are set to, click on the **Show Details >>** button and it will display the full dialog box as above.

Download at Boykma.Com

As this is our first time, we have to enter all the details. The **User Name** and **Password** are what we specified in the Repository Assistant for the workspace owner, and the **Connection details** are the **Host**, **Port**, and **Service Name** we specified when we used the Database Configuration Assistant to create our database. We'll enter **acmeowb** as the user name and **acmedw** as the service name.

Use the **Workspace Management** button to invoke **Repository Assistant** from the **Design Center Logon** dialog box. This will allow us to configure our workspace for additional users or create a new workspace if needed.

The main **Design Center** window will be displayed next upon a successful log on. An example is shown in the following image, which depicts the default appearance of the **Design Center** with the entries expanded in the **Project Explorer** and **Connection Explorer** windows:



A project called **MY_PROJECT** appears, which is the default project that the Warehouse Builder will create in every workspace.

Download at Boykma.Com

We referred to **MY_PROJECT** back when we discussed the final results screen of the Repository Assistant, which showed this project name even though we hadn't specified one.

Before discussing the project in more detail, let's talk about the three windows in the main **Design Center** screen. They are as follows:

- **Project Explorer**
- **Connection Explorer**
- **Global Explorer**

The **Project Explorer** window is where we will work on the objects that we are going to design for our data warehouse. It has nodes for each of the design objects we'll be able to create. It is not necessary to make use of every one of them for every data warehouse we design, but the number of options available shows the flexibility of the tool. The objects we need will depend on what we have to work with in our particular situation. In our analysis earlier, we determined that we have to retrieve data from a database where it is stored.

So, we will need to design an object under the **Databases** node to model that source database. If we expand the **Databases** node in the tree, we will notice that it includes both **Oracle** and **Non-Oracle** databases. We are not restricted to interacting with just Oracle in Warehouse Builder, which is one of its strengths. We will also talk about pulling data from a flat file, in which case we would define an object under the **Files** node. If our organization was running one of the applications listed under the **Applications** node (which includes **Oracle E-Business Suite**, **PeopleSoft**, **Siebel**, or **SAP**) and we wanted to pull data from it, we'd design an object under the **Applications** node.

The **Project Explorer** isn't just for defining our source data, it also holds information about targets. Later on when we start defining our target data warehouse structure, we will revisit this topic to design our database to hold our data warehouse. So the **Project Explorer** defines both the sources of our data and the targets, but we also need to define how to connect to them. This is what the **Connection Explorer** is for.

The **Connection Explorer** is where the connections are defined to our various objects in the **Project Explorer**. The workspace has to know how to connect to the various databases, files, and applications we may have defined in our **Project Explorer**. As we begin creating modules in the **Project Explorer**, it will ask for connection information and this information will be stored and be accessible from the **Connection Explorer** window. Connection information can also be created explicitly from within the **Connection Explorer**.

Download at Boykma.Com

Multiple projects can be defined in **Project Explorer**, but connection information is not displayed projectwise in the **Connection Explorer**. Connections are applicable for the entire workspace, and not just the project we are working on.

There are some objects that are common to all projects in a workspace. The **Global Explorer** is used to manage these objects. It includes objects such as **Public Transformations** or **Public Data Rules**. A **transformation** is a function, procedure, or package defined in the database in Oracle's procedural SQL language called PL/SQL. **Data rules** are rules that can be implemented to enforce certain formats in our data.

Importing/defining source metadata

Now that we've been introduced to the **Design Center**, it's time to make use of it to import or define our source metadata. **Metadata** is data that describes our data. We are going to tell the Warehouse Builder what our source data looks like and where it is located, so that it can build the code necessary to retrieve that data when we design and run mappings to populate our data warehouse. The metadata is represented in the Warehouse Builder as objects corresponding to the type of the source object. So if we're representing tables in a database, we will have tables defined in the Warehouse Builder.

We have a couple of options for defining the source metadata. We can manually input the definitions into **Design Center Project Explorer** ourselves, or we can choose to have the Warehouse Builder automatically import the descriptions of our data for us. As we like having the computer do the work for us whenever possible, we will choose the second option whenever we can.

We need to be clear about the difference between importing or defining the metadata for our sources and loading the actual data as it can be confusing. At this stage, we are just importing or defining the definitions of our objects. (Metadata, or data about data, is information that tells us what the data looks like, column names, data types, and so on.) Later when we implement our targets and actually create a mapping between the source and the target and deploy it, we will be loading the actual data.

Download at Boykma.Com

Creating a project

The very first thing we have to do in **Design Center** is make sure we have a project defined that will hold all of our work. In the last image, we saw a depiction of the **Design Center** as it appears when we first log on. Launch the **Design Center** now if you haven't already and we'll start working with it.

We can choose to use the default **My Project** project that was created for us, or create another new one. We are just going to use this default project as the Warehouse Builder was nice enough to create it for us. But, oh, that name is so boring. Let's give it a new name that is more appropriate for our company project. So right-click on the project name in the **Project Explorer** and select **Rename** from the resulting pop-up menu. Alternatively, we can select the project name, then click on the **Edit** menu entry, and then on **Rename**. In either case, the name will be highlighted and turned to italics and we'll be able to use the keyboard to type a new name. We'll name the project `ACME_DW_PROJECT`.

If we wanted to create a new project, we would select **New...** either from the pop-up menu or from the **Design** drop-down menu. We can have any number of projects defined, but can work on only one at a time. There's a high possibility that we might be building more than one data warehouse at a time, and we could have a separate project defined for each.

Creating a module

Creating a project is the first step. But before we can define or import a source data definition, we must create a module to hold it. A **module** is an object in the **Design Center** that acts as a storage location for the various definitions and helps us logically group them. There are **Files** modules that contain file definitions and **Databases** modules that contain the database definitions. These **Databases** modules are organized as **Oracle** modules and **Non-Oracle** modules. Those are the main modules we're going to be concerned with here. We have to create an Oracle module for the **ACME_WS_ORDERS** database for the web site orders, and a non-Oracle module for the **ACME_POS** SQL Server database. We'll create the Oracle module first because it is the simplest. After that we'll create the module for the SQL Server database, which will involve a few more steps because Oracle has to communicate with the SQL Server database.

Download at Boykma.Com

Creating an Oracle Database module

To create an **Oracle Database** module, right-click on the **Databases | Oracle** node in the **Project Explorer** of Warehouse Builder and select **New...** from the pop-up menu. The first screen that will appear is the **Welcome** screen, so just click on the **Next** button to continue. Then we have the following two steps:

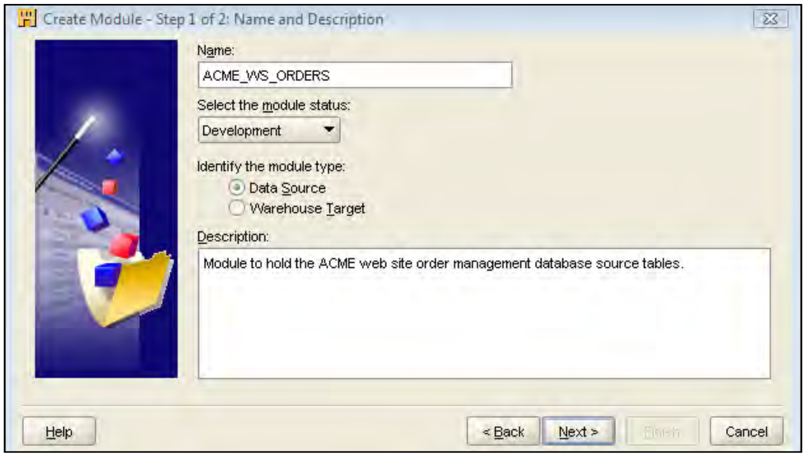
1. In this step we give our new module a name, a status, and a description that is optional. We do the following in this step:
   - On the next window after the **Welcome** screen, type in a name for the module.

     The name should reflect the name of the source database for consistency and ease of matching the module to the source database later. We're going to name our module **ACME_WS_ORDERS**, which is the name of ACME's Web Site Orders Oracle database as we discovered earlier when doing our analysis of the existing systems.
   - The module status is a way of associating our module with a particular phase of the process, and we'll leave it at **Development**.
   - For the description, just enter any text that helps describe the source.
   - We also have to indicate the module type—whether it is for a data source or a warehouse target. As this is an Oracle database module, we can select either one. As we'll see in the next section, this option won't be available for a SQL Server database using ODBC.

In general, OWB supports Oracle natively as a target. But in Chapter 4 of the *User's Guide*, the Warehouse Builder documentation does mention that other databases and even spreadsheets can be targets. However, it also goes on to say that to load data into third-party databases or spreadsheets, first deploy to a comma-delimited or XML formatted flat file. So, the support is not quite the same, which is why **Non-Oracle** Database modules only have **Data Source** available as an option.

We'll select **Data Source** and our screen should now look similar to the following:

Download at Boykma.Com


Create Module - Step 1 of 2: Name and Description

Name: ACME_WS_ORDERS

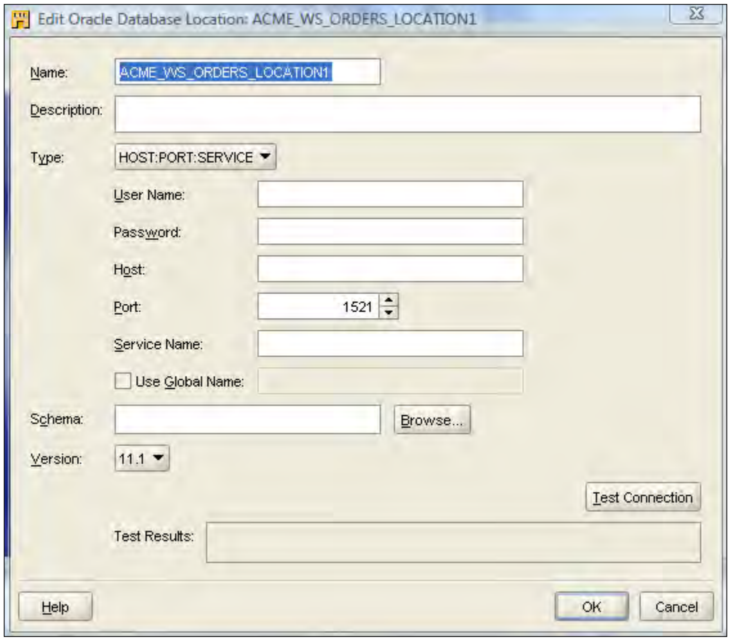
Select the module status: Development

Identify the module type:
(● Data Source) ○ Warehouse Target

Description: Module to hold the ACME web site order management database source tables.

Click on the **Next** button to proceed to defining the connection.

2. In this step, we define the connection information for Warehouse Builder so that it knows how to connect to the source. We do that in the next screen using the following steps:
   - The screen starts by suggesting a connection name based on the name we gave the module. Click on the **Edit** button beside the **Name** field to fill in the details. This will display the following screen:


Edit Oracle Database Location: ACME_WS_ORDERS_LOCATION1

Name: ACME_WS_ORDERS_LOCATION1

Description:

Type: HOST:PORT:SERVICE

User Name:

Password:

Host:

Port: 1521

Service Name:

☐ Use Global Name:

Schema:

Version: 11.1

Test Results:

Download at Boykma.Com

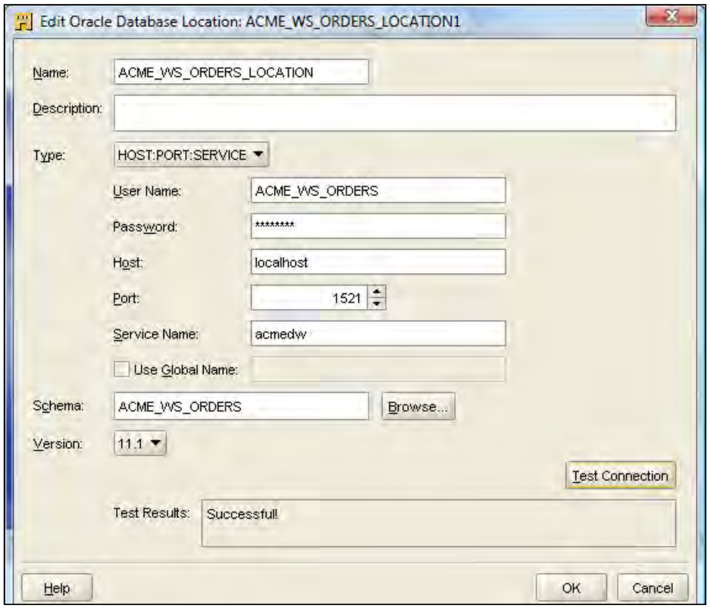
The name it suggested for us is **ACME_WS_ORDERS_LOCATION1**. That's a fine name, except that 1 is on the end, so let's just remove it.

- For the connection details, we're going to enter **User Name**, `acme_ws_orders`, and the **Password** that was given to us by the DBA for the web site orders system. When we type in the username and move to the next field, the schema field will be automatically populated with the username.

> If you are using the scripts downloaded from our web site, the default password used is **acme1234** for the `acme_ws_orders` user.

- Enter the **Host** where the Oracle database resides and contains the `acme_ws_orders` schema, which is `localhost` as we're running everything on one system.
- The **Port** that the listener is listening on is **1521** so leave it as the default. Enter **acmedw** as the **Service Name** for the Oracle database. The schema we'll be connecting to has been automatically filled in for us.
- One final step is to make sure the version of the Oracle database is set correctly. The **Version** we're working with is **11.1**, the most recent and the default.

We should now have a screen that looks similar to the following:



Edit Oracle Database Location: ACME_WS_ORDERS_LOCATION1

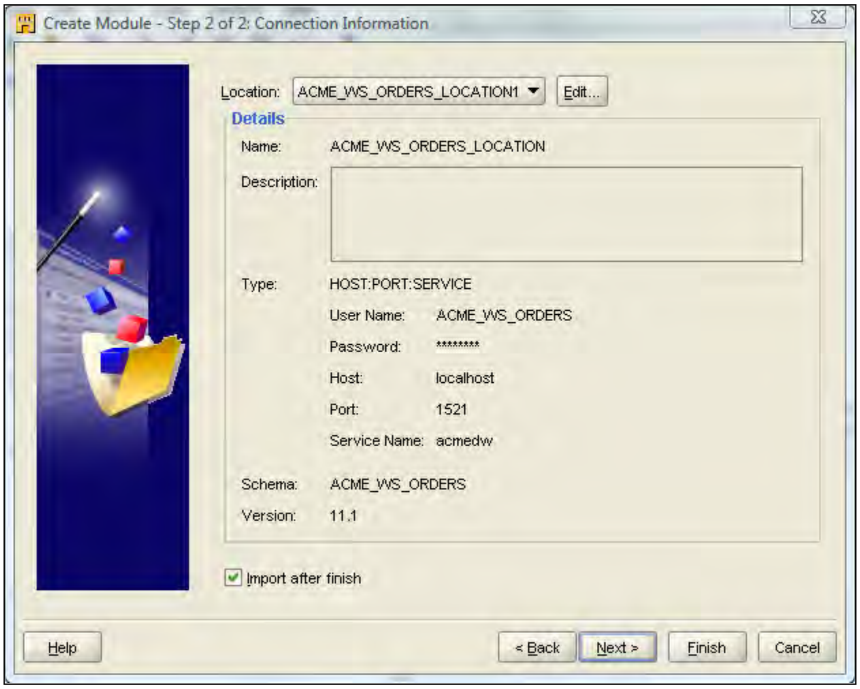
Name: ACME_WS_ORDERS_LOCATION
Description:
Type: HOST:PORT:SERVICE
User Name: ACME_WS_ORDERS
Password: ********
Host: localhost
Port: 1521
Service Name: acmedw
Use Global Name:
Schema: ACME_WS_ORDERS
Version: 11.1
Test Results: Successful!

Download at Boykma.Com

- Press the **Test Connection** button and if everything is OK, we'll see a **Successful!** message as shown in the previous screenshot. If not, it will display the error(s) in that window for us so that we can debug the problem.

If we do get any errors, it's a good idea to become intimately familiar with the error manual in the Oracle documentation, which can be found at `http://download.oracle.com/docs/cd/B28359_01/server.111/b28278/toc.htm`. The errors will usually start with the three characters `ORA`, followed by a hyphen and then the error number. We can use all this to look up an error in the error manual to get more information. Google, Yahoo, or any Internet search engine can also be our friend here as (unfortunately) some of the errors, even after we look them up in the error manual, are not exact about the cause of the problem. Usually, searching for the error message string on the Internet can turn up others who have encountered the same issue and explanations of what to do about it.

- Click on the **OK** button to proceed, even if an error was reported when we clicked on the **Test Connection** button. Now we will be back at the **Step 2** window where all the connection results will be filled in and it will be ready to create the module as shown here:



Location: ACME_WS_ORDERS_LOCATION1 Edit...

Details

Name: ACME_WS_ORDERS_LOCATION

Description:

Type: HOST:PORT:SERVICE

User Name: ACME_WS_ORDERS

Password: ********

Host: localhost

Port: 1521

Service Name: acmedw

Schema: ACME_WS_ORDERS

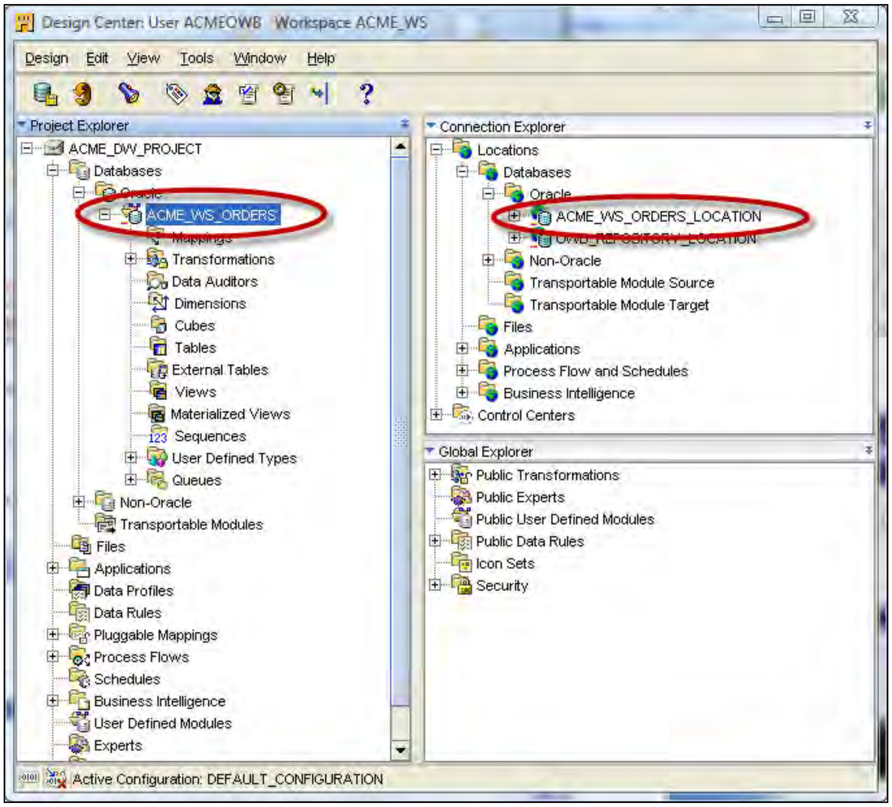
Version: 11.1

☑ Import after finish

Download at Boykma.Com

- The **Import after finish** checkbox will be checked by default. But we're going to uncheck it because we're going to move on and create a module for the SQL Server database before we import any metadata. So, uncheck the box and click on the **Finish** button.

We are now back at the main Warehouse Builder interface and we can see that it has added our new module under the **Databases | Oracle** node in the **Project Explorer**. If we expand the **Locations | Databases | Oracle** module in the **Connection Explorer**, we'll see our location **ACME_WS_ORDERS_LOCATION** listed as shown in the following image. We just defined this location as a part of the process of creating the module.

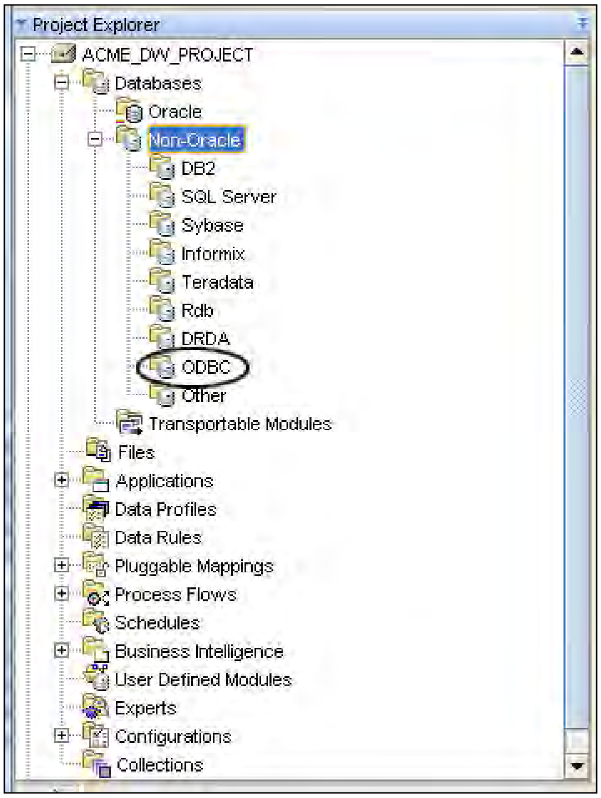


Even if we had an error during the previous process of creating this connection, we would still see these entries created. If we could fix whatever caused the error, we'd then have a valid working connection without having to go back through the wizard to create it again.

Download at Boykma.Com

Creating a SQL Server database module

Now that we have our module created for the Oracle database, let's create one for the SQL Server POS transactional database: **ACME_POS**. First, let's talk in brief about the external database connections in Oracle. If we expand the **Databases** node in the **Project Explorer**, we'll see **Oracle**, **Non-Oracle**, and **Transportable Modules**. The **Non-Oracle** entry has a little plus sign next to it that indicates there are further subnodes under it, so let's expand that to see what they are. In the following image, a number of databases are listed, including the one that says just **ODBC**:



The POS transactional database is a Microsoft **SQL Server** database and we'll notice that it is one of the databases listed by name under the **Non-Oracle** databases. We might think this is where we're going to create our source module for this import, but no.

Download at Boykma.Com

Oracle makes use of **Oracle Heterogeneous Services** to make connections to other databases. This is a feature that makes a non-Oracle database appear as a remote Oracle database server. There are two components to make this work—the heterogeneous service that comes by default with the Oracle database and a separate agent that runs independently of the database. The **agent** facilitates the communication with the external non-Oracle database. That agent can take one of these two forms:

- A **transparent gateway agent** that is tailored specifically to the database being accessed
- A **generic connectivity agent** that is included with the Oracle Database and which can be used for any external database

The transparent gateway agents must be purchased and installed separately from the Oracle Database, and then configured to support the communication with the external database. They are provided for heavy-duty applications that do a large amount of external communication with other databases. This is much more than we need for this application and as we have to pay extra for it, we are going to stick with the generic connectivity agent that comes free with the Oracle Database. It is a low-end solution that makes use of ODBC or OLE-DB drivers for accessing the external database. **ODBC** (**Open Database Connectivity**) is a standard interface for accessing database systems and is platform and database independent. **OLE-DB** (**Object Linking** and **Embedding-Database**) is a Microsoft-provided programming interface that extends the capability provided by ODBC to add support for other types of non-relational data stores that do not implement SQL such as spreadsheets.

There is a significant differences between transparent gateways and the generic connectivity agent. The generic connectivity agent is restricted to the features of ODBC or OLE-DB and is very generic as a result. Transparent gateways are specifically tailored to the non-Oracle database and support a much wider range of database access features. As a result, one aspect to consider is how extensive our access to the other database will be from Oracle and how many of Oracle's features we'll need to use. One of the benefits of Oracle Heterogeneous Services is that it allows us to make use of a non-Oracle database as if it were an Oracle database. This includes features of Oracle database access that aren't available from the database we are accessing (such as using PL/SQL to access the database). The generic connectivity agent is limited in some of the features it allows when accessing another non-Oracle database, and this factor may depend on whether we need these features or not.

Download at Boykma.Com

Refer to the documentation to make a decision about which would be an appropriate choice of the agent in your case. The *Heterogeneous Connectivity Administrator's Guide* can be found at the following URL: `http://download.oracle.com/docs/cd/B28359_01/server.111/b28277/toc.htm`. The *Gateway for ODBC User's Guide* documentation can be found here: `http://download.oracle.com/docs/cd/B28359_01/gateways.111/e10311/toc.htm`.

The agent we choose will determine which of the nodes under the **Databases | Non-Oracle** node will be used to create our SQL Server database module.

The individually named database nodes are used if we're using a transparent gateway agent tailored for that database. The ODBC node is the one we use for any database connections using the generic connectivity agent.

Now that we've decided to use the generic connectivity solution, we need to create an **ODBC** module in Warehouse Builder to hold our definitions of source data for the POS transactional database. As this is a database we're using for the source, the module will be created under the **Databases | Non-Oracle** node in the Warehouse Builder and not under the **Databases | Oracle** node as it is not an Oracle database. Expanding the **Databases** node and then the **Non-Oracle** node as shown in the previous image, we see that there is an **ODBC** node available. It is under this node that we will create our module for the source definitions for the POS transactional database.

However, there is one problem—because this is a non-Oracle database we're connecting to, we have to provide information to our Oracle database so that it knows how to connect. Warehouse Builder uses the underlying Oracle database to make the connection. So this information must be provided before we define our module and location in the Warehouse Builder. In the following section, we will go through the steps to define our connection to the SQL Server database named `ACME_POS`. We're going to depart briefly from Warehouse Builder-specific topics here, but only because this is necessary for us to continue in the Warehouse Builder.

Download at Boykma.Com

If you are following along with each of these steps and want to create the `ACME_POS` database, you can run the scripts that have been provided in SQL Server to create the database and tables. They are available for download from the Packt web site at `http://www.packtpub.com/files/code/5746_Code.zip`. Microsoft SQL Server 2008 Express was used for this book to generate the scripts because it is available free of charge. It is available from Microsoft's web site at `http://www.microsoft.com/express`. It is available without charge and provides all the SQL Server functionality we'll need for this book.

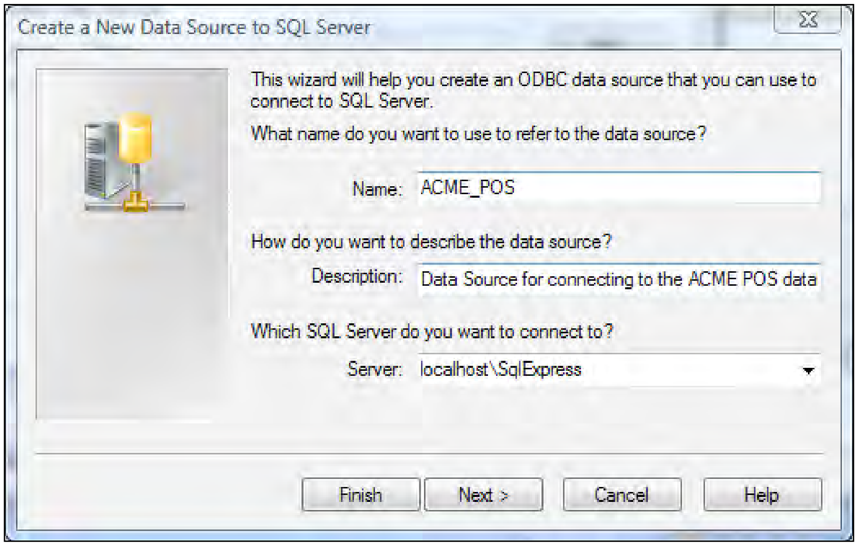
Creating a SQL Server database connection

The first step that is required in making use of Oracle Heterogeneous Services to access a non-Oracle database using the generic connectivity agent is to create an ODBC connection. We do that by setting up a system **DSN** (**Data Source Name**). A DSN is the name you give to an ODBC connection. An **ODBC connection** defines which driver to use and other physical connection details that are needed to make a connection to a database. On Microsoft Windows, we configure DSNs in the **ODBC Data Source Administrator**. The following are the steps for configuring DSN:

1. You can access this application by navigating through the **Start | Control Panel | Administrative Tools** menu. The application is called **Data Sources (ODBC)**.
2. In **ODBC Data Source Administrator**, click on the **System DSN** tab, and then click on the **Add** button to add a new system DSN.
3. The first screen asks you to select which driver you want to use for your data source. ODBC drivers are specific to a database, so you have to use the one that is defined for accessing a SQL Server database. Scroll down the list until you see the **SQL Server** entry and click on it. Now click on the **Finish** button.

   This will take you to the screens that create an ODBC data source for connecting to SQL Server. Each ODBC driver requires a different configuration depending on the database it is connecting to.
4. For SQL Server, the first screen will ask you for a **Name**, **Description**, and the host on which the SQL Server database is located. For clarity, let's name our DSN **ACME_POS** after the database, enter **Data Source for connecting to the ACME POS database** for the description, and **localhost\SqlExpress** for the hostname in the **Server** field as illustrated in the following image:

Download at Boykma.Com


Create a New Data Source to SQL Server

This wizard will help you create an ODBC data source that you can use to connect to SQL Server.

What name do you want to use to refer to the data source?

Name: ACME_POS

How do you want to describe the data source?

Description: Data Source for connecting to the ACME POS data

Which SQL Server do you want to connect to?

Server: localhost\SqlExpress

We're entering **localhost** here because in working through the examples in this book, all the required applications/databases are installed on one system. In actual business environments, the databases we are going to be using for source databases will most likely be located on other computers elsewhere on the network. We will enter the hostname for the other machine on which the SQL Server database is located.

5. Click on the **Next** button to proceed.

Notice **\SqlExpress** at the end of the hostname. This is required because SQL Express is installed as what is called a **named instance**, which basically requires that the name be included with the hostname for it to be found successfully.

6. In the next screen we will specify the authentication method to use to connect to the database. We have two options here. We can use **Windows NT Authentication using the network login ID**. (SQL Server will use the network or local machine login ID of the user connected at that time.) Alternatively, we can use **SQL Server authentication using a login ID and password** provided to us. The ACME DBA in charge of the `ACME_POS` database has kindly set up a username for us to access the `ACME_POS` database for importing definitions and data. He's given that user read permission on the tables in the database. The username is **acme_dw_user**. So we will use the second option.

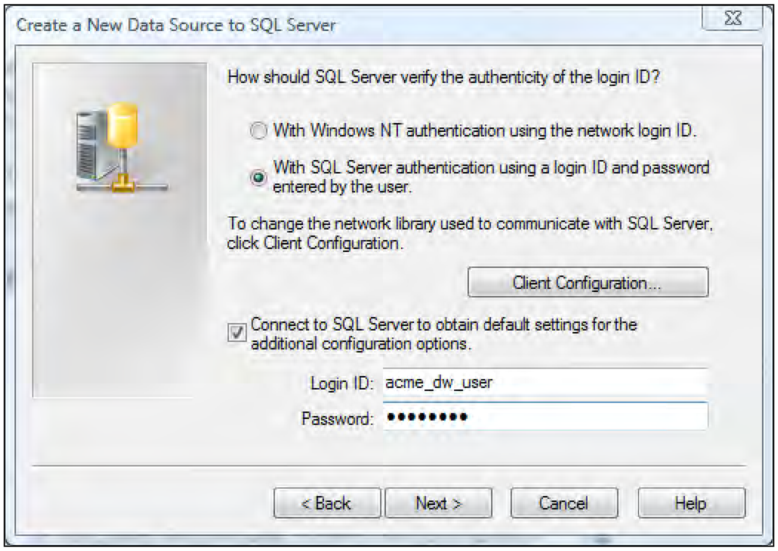
Download at Boykma.Com

> The scripts that are provided with this book are available for download and can be used to set up the SQL Server database to work through the examples in the book. This database uses the names that are provided here for the database and user.

7. There is a checkbox at the bottom of the screen to check off and have the new data source wizard connect to the SQL Server to obtain additional information. We are going to check that box if it is not checked by default, and enter the username and password provided by the `ACME_POS` DBA.

> An important item to note here is that this username and password are used only by the DSN creation application to access the database for some additional configuration items during the DSN setup process. This username and password will not be used by any application that subsequently uses our ODBC DSN to connect to the SQL Server. We will provide those connection details in a moment when we get to define the connection in the Warehouse Builder.

8. This is how our screen looks and we will click on **Next** to continue:


Create a New Data Source to SQL Server

How should SQL Server verify the authenticity of the login ID?

( ) With Windows NT authentication using the network login ID.

[✓ With SQL Server authentication using a login ID and password entered by the user.]

To change the network library used to communicate with SQL Server, click Client Configuration.

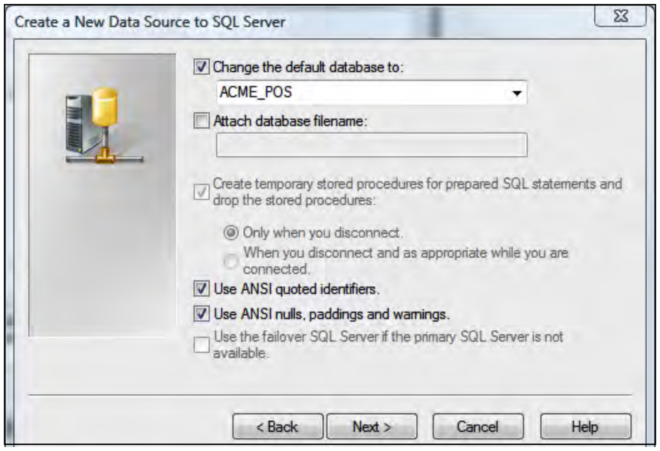
[✓] Connect to SQL Server to obtain default settings for the additional configuration options.

Login ID: acme_dw_user

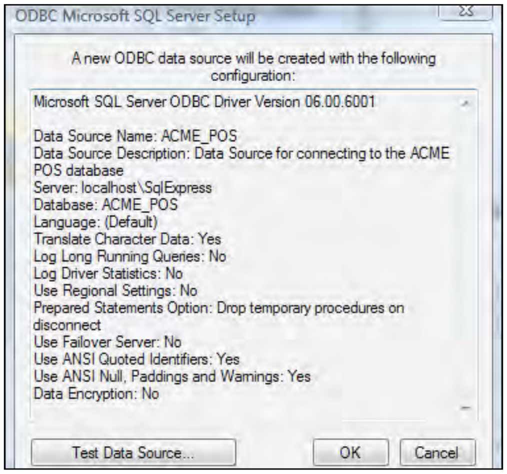
Password: ••••••••

Download at Boykma.Com

9. In the next screen, the primary item we want to verify is whether the default database is listed as **ACME_POS** so that when we use the ODBC connection it is connected to the correct database. It's quite possible for the username provided to have access to more than one database on the SQL Server instance if more than one exists. If the correct database is not showing, then check the box beside the database name and select the correct database as shown in the following screenshot:


Create a New Data Source to SQL Server
☑ Change the default database to:
ACME_POS
☐ Attach database filename:
☑ Create temporary stored procedures for prepared SQL statements and drop the stored procedures:
◉ Only when you disconnect.
○ When you disconnect and as appropriate while you are connected.
☑ Use ANSI quoted identifiers.
☑ Use ANSI nulls, paddings and warnings.
☐ Use the failover SQL Server if the primary SQL Server is not available.
< Back
Next >
Cancel
Help

10. Leave all the other options set as they are and click on the **Next** button to continue.
11. The next screen is full of configuration options that we should just leave set to the defaults and click on the **Finish** button to complete the process. This will present us with the final summary screen of the ODBC connection details as shown in the following screenshot:



A new ODBC data source will be created with the following configuration:

```
Microsoft SQL Server ODBC Driver Version 06.00.6001

Data Source Name: ACME_POS
Data Source Description: Data Source for connecting to the ACME
POS database
Server: localhost\SqlExpress
Database: ACME_POS
Language: (Default)
Translate Character Data: Yes
Log Long Running Queries: No
Log Driver Statistics: No
Use Regional Settings: No
Prepared Statements Option: Drop temporary procedures on
disconnect
Use Failover Server: No
Use ANSI Quoted Identifiers: Yes
Use ANSI Null, Paddings and Warnings: Yes
Data Encryption: No
```

Download at Boykma.Com

If we want, we can test the newly created data source right here. If we click on the **Test Data Source...** button, it will make a connection to the database and return a screen indicating success or failure. Click on the **OK** button on this screen and the ODBC connection will be created. It will now appear on the **System DSN** tab of the **ODBC Data Source Administrator**.

Configure Oracle to connect to SQL Server

Let's move on to the next step in the process of getting Oracle Heterogeneous Services to connect to our SQL Server database. We will configure Oracle now that we have our ODBC connection created. The following are the two steps involved here:

1. Create a heterogeneous service configuration file.
2. Edit the `listener.ora` file.

Creating a heterogeneous service configuration file

We will be creating a heterogeneous service configuration file in the `ORACLE_HOME\hs\admin` folder. Just substitute your applicable `ORACLE_HOME` location. The following are the steps to create this file:

1. Open **Windows Explorer** and navigate to this folder.

   There is a sample **init** file that Oracle has been kind enough to supply us with. We can easily modify this file to suit our purpose. It is a plain-text file, so we can use any text editor to edit it.

2. Let's open the file named `initdg4odbc.ora` in our favorite text editor, or Windows Notepad if we don't have any other text editor.

   This is the default init file for using ODBC connections. The contents will basically look like the following:

```
# This is a sample agent init file that contains the HS parameters
#that are needed for the Database Gateway for ODBC
#
# HS init parameters
#
HS_FDS_CONNECT_INFO = <odbc data_source_name>
HS_FDS_TRACE_LEVEL = <trace_level>
#
# Environment variables required for the non-Oracle system
#
#set <envvar>=<value>
```

Download at Boykma.Com

The lines that begin with # are comment lines and will be ignored. The two lines we're interested in are the ones that are in bold in the code we just saw.

3. The `HS_FDS_CONNECT_INFO` line is where we specify the ODBC DSN that we just created in the previous section. So replace the `<odbc data_source_name>` string with the name of the Data Source, which is (unless you changed it from what was suggested) `ACME_POS`.

4. The `HS_FDS_TRACE_LEVEL` line is for setting a trace level for the connection. The trace level determines how much detail gets logged by the service and it is OK to set the default as `0` (zero).

> To read more about what this entry's purpose is, refer to the *Oracle Database Heterogeneous Connectivity Administrator's Guide 11g Release 1* at the following URL: `http://download.oracle.com/docs/cd/B28359_01/server.111/b28277/toc.htm`.

Having made those changes, our file should now look like the following:

```
# This is a sample agent init file that contains the HS
# parameters that are
# needed for the Database Gateway for ODBC

#
# HS init parameters
#
HS_FDS_CONNECT_INFO = ACME_POS
HS_FDS_TRACE_LEVEL = 0


#
# Environment variables required for the non-Oracle system
#
#set <envvar>=<value>
```

5. Now we will save the file with a new name and will be careful not to overwrite the default file. We'll give it a name that begins with `init` and ends with `.ora`, and contains a name in the middle that is descriptive and does not contain spaces or special characters. Let's save it as `initacmepos.ora`.

Leave out the underscore character as we're not allowed to use special characters. We might think it's just a filename and it is certainly allowed to use an underscore in the filename. However, this part of the file name must be used in the next step for a purpose that does not allow special characters to be used.

Download at Boykma.Com

Editing the listener.ora file

Now we're going to add a SID to our `listener.ora` file. When we configured the listener back in *Chapter 1*, it created a `listener.ora` file in `ORACLE_HOME\network\admin`. The steps for this are:

1. Load the `listener.ora` file into a text editor (or Notepad). Add the following lines to the file:

   ```
   SID_LIST_LISTENER=
     (SID_LIST=
         (SID_DESC=
            (SID_NAME=acmepos)
            (ORACLE_HOME=D:\app\bob\product\11.1.0\db_1)
            (PROGRAM=dg4odbc)
         )
     )
   ```

   There is a sample `listener.ora` file called `listener.ora.sample`, which is provided for us in the `ORACLE_HOME\hs\admin` folder. It contains the above lines that can be cut and pasted into our actual `listener.ora`. We just need to correct `SID_NAME` to `acmepos`.

   For `SID_NAME`, we have to specify the name we used as part of the name of our init file in the previous step. This is why no special characters were allowed because this name will become the SID for our database connection and SIDs cannot have special characters. However, don't include the `init` or `.ora` from the name of this file.

   In the `PROGRAM` entry, we will specify the agent that will handle the connectivity for us and the name of the generic connectivity agent program supplied with the Oracle Database 11*g* is `dg4odbc`. For `ORACLE_HOME`, you will substitute your particular `ORACLE_HOME` location, which will be different unless your username is also `bob` and you installed Oracle using the default naming convention on the `D` drive .

Download at Boykma.Com

An important tip about the PROGRAM name

Make sure you use the correct name for `PROGRAM` for your version of the database. In versions prior to 11*g*, the generic connectivity agent name was `hsodbc`. However, in Oracle Database 11*g*, it is known as `dg4odbc`. If we use the wrong name for `PROGRAM`, or misspell it, we will get a strange error message when we try to define our connection information using this external link. Do not make the mistake of referring to the example in Chapter 13 of the *OWB Users Guide* (`http://download.oracle.com/docs/cd/B28359_01/owb.111/b31278/toc.htm`) about connecting to SQL Server. They list a nice example of how to edit the `listener.ora` file, but show `hsodbc` as the `PROGRAM` name. If we look closer, we'll see that `ORACLE_HOME` they reference is an Oracle 10*g* home even though this example is in the Warehouse Builder 11*g* Users Guide!

There may already be a `SID_LIST_LISTENER` entry in the `listener.ora` file. If so, just add the `SID_DESC` section above the existing `SID_LIST_LISTENER` entry. By studying that entry, you can see the syntax for how the `SID_DESC` sections are listed; so just follow the same convention.

2. After we save the `listener.ora` file, we must restart the listener for the change to take effect. We can restart it by navigating to **Start | Control Panel | Administrative Tools** and then clicking on **Services**. Now, scroll down until you see the service for your database listener, which will be named starting with **Oracle** and ending in **TNSListener**. It will contain **ORACLE_HOME—OracleOraDb11g_home1TNSListener**. Now right-click on it and select **Restart**.

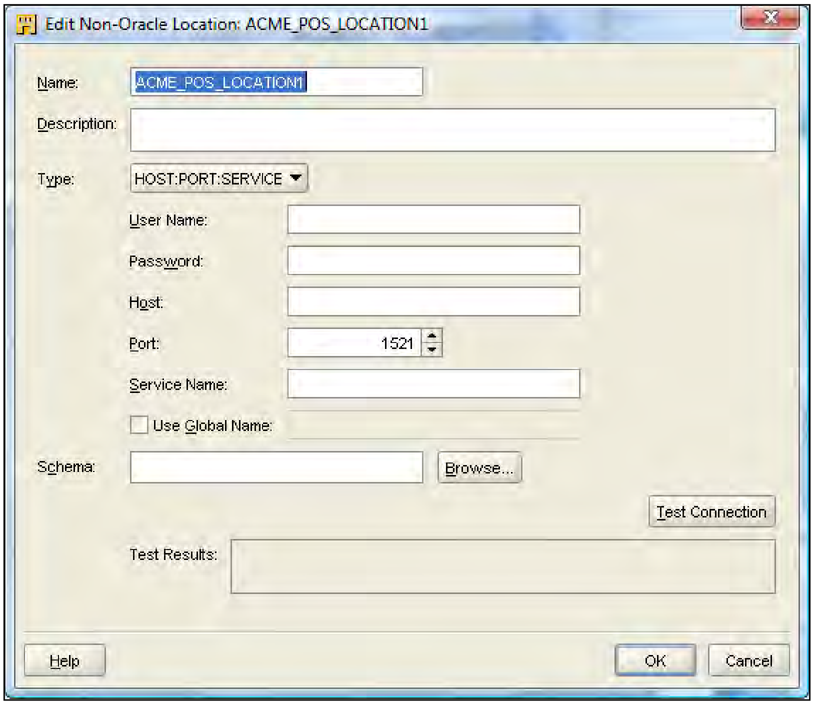
Creating the Warehouse Builder ODBC module for SQL Server

Now that we have defined our source SQL Server database connection information in Oracle, we are done with our foray into non-Warehouse Builder-specific topics. We will get back to the main topic of creating the module and location in the Warehouse Builder. This process is very similar to creating a module for an Oracle database as we just did. There are some slight differences in a couple of screens, which we'll point out as we go along. The steps to create an ODBC module and location in Warehouse Builder are:

1. Right-click on the **ODBC** node in the **Project Explorer** of **Design Center**, and select **New...** from the pop-up menu. The first screen that will appear is the **Welcome** screen, so just click on the **Next** button to continue.
2. The screen with the label **Step 1** is where we provide a name as we did for the Oracle module. We're going to name this ODBC module `ACME_POS`, which is the name of ACME's POS transactional database in SQL Server as we discovered earlier when analyzing the existing systems.

Download at Boykma.Com

3. We'll leave **module status** set to **Development**.
4. One difference we'll see from the **Step 1** screen earlier for an Oracle module is that there is no option to choose the type of module. It is a **Data Source** module by default. Click on the **Next** button to proceed to defining the connection.
5. The next screen labeled **Step 2** is for the connection just as earlier. We'll click on the **Edit** button beside the name to fill in the details. This will display the following screen:


Edit Non-Oracle Location: ACME_POS_LOCATION1

Name: ACME_POS_LOCATION1
Description:
Type: HOST:PORT:SERVICE
User Name:
Password:
Host:
Port: 1521
Service Name:
Use Global Name:
Schema:
Test Results:

We'll remove the **1** as we did for the Oracle connection.

6. For the connection details, we will enter the **User Name** as `"acme_dw_user"`, and **Password**, which was given to us by the DBA for the transactional system.

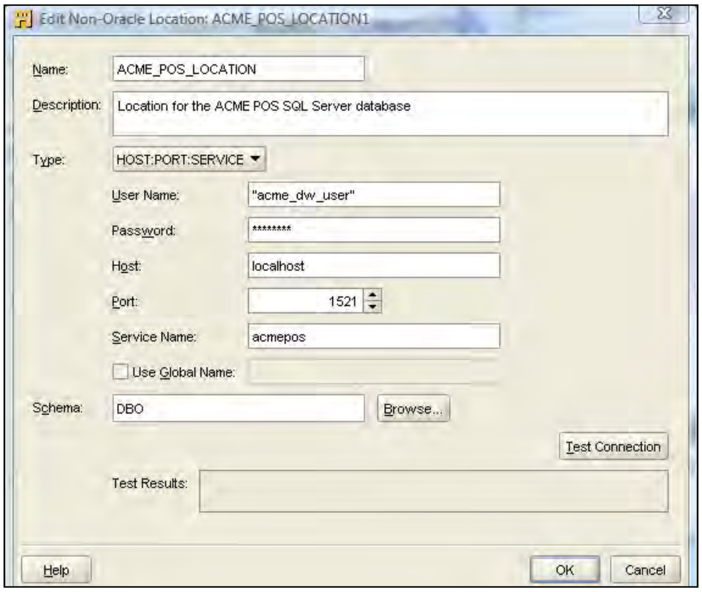
We have to make sure that both *username and password* are *enclosed in double quotes*. The Oracle database will automatically make them uppercase if we don't, and the SQL Server database does not like that. So, if we get a username and/or password incorrect error, we'll double-check that we enclosed them in double quotes; yes, even the password. The double quotes in the password will appear as asterisks like the rest of the password, but make sure to put them in there.

Download at Boykma.Com

7. Enter the **Host** where the Oracle database resides and where we configured the heterogeneous services. It is **localhost** as we're running everything on the same system.

> Here we might think that we have to enter the hostname of where the SQL Server database resides, as we're entering connection information to connect to it. Remember that although we're using Oracle Heterogeneous Services to make that connection for us and have already gone through the steps to configure it in the listener to connect to the ODBC DSN, where the actual connection information for the SQL Server database is specified. This means what we need to specify here is the connection information for the SID that we configured earlier as if we're connecting to an Oracle database. In reality, it will actually be connecting to the SQL Server database.

8. The **Port** the listener is listening on is **1521**, so leave it as the default. Enter the **Service Name** that we configured in the previous section in the listener for the generic connectivity dg4odbc agent—**acmepos**.
9. Finally, enter the schema we'll be connecting to. For SQL Server, the owner of most databases is referred to internally as **DBO** and so this is what we're going to put here. We should now have a screen that looks similar to the following:


Edit Non-Oracle Location: ACME_POS_LOCATION1

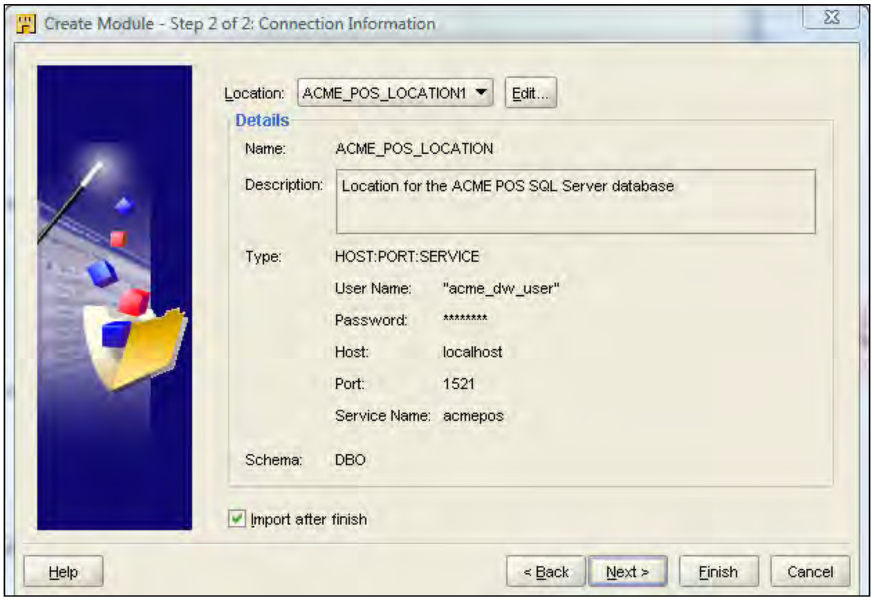
Name: ACME_POS_LOCATION
Description: Location for the ACME POS SQL Server database
Type: HOST:PORT:SERVICE
User Name: "acme_dw_user"
Password: ********
Host: localhost
Port: 1521
Service Name: acmepos
Use Global Name:
Schema: DBO
Test Results:

10. We can click on the **Test Connection** button to make sure everything is working properly and the results will be displayed in the **Test Results** field.

Download at Boykma.Com

> This is where we may encounter an error if the PROGRAM name is incorrect in the listener.ora file. Here's such an example of an error and you can see how unhelpful these error messages can be: **ORA-28545: error diagnosed by Net8 when connecting to an agent Unable to retrieve text of NETWORK/NCR message 65535**. Even if we search the Internet and documentation for solutions, many suggestions will mention hsodbc, and not dg4odbc. The solution to this particular error actually refers to the PROGRAM name. In this case, the initdg4odbc.ora and the listener.ora.sample example files in the ORACLE_HOME\hs\admin folder clearly say dg4odbc and not hsodbc. Those who work a lot with Oracle 10*g* may (out of habit) have used hsodbc, never once thinking about double-checking whether that was correct or not. The moral of the story: Use the example files as a starting point!

11. Click on the **OK** button to proceed even if there was an error reported when we clicked on the **Test Connection** button. We will be back at the **Step 2** window with all the connection results now filled in and we will be ready to create the module as shown here:


Create Module - Step 2 of 2: Connection Information

Location: ACME_POS_LOCATION1 Edit...

Details

Name: ACME_POS_LOCATION

Description: Location for the ACME POS SQL Server database

Type: HOST:PORT:SERVICE

User Name: "acme_dw_user"

Password: ********

Host: localhost

Port: 1521

Service Name: acmepos

Schema: DBO

✓ Import after finish

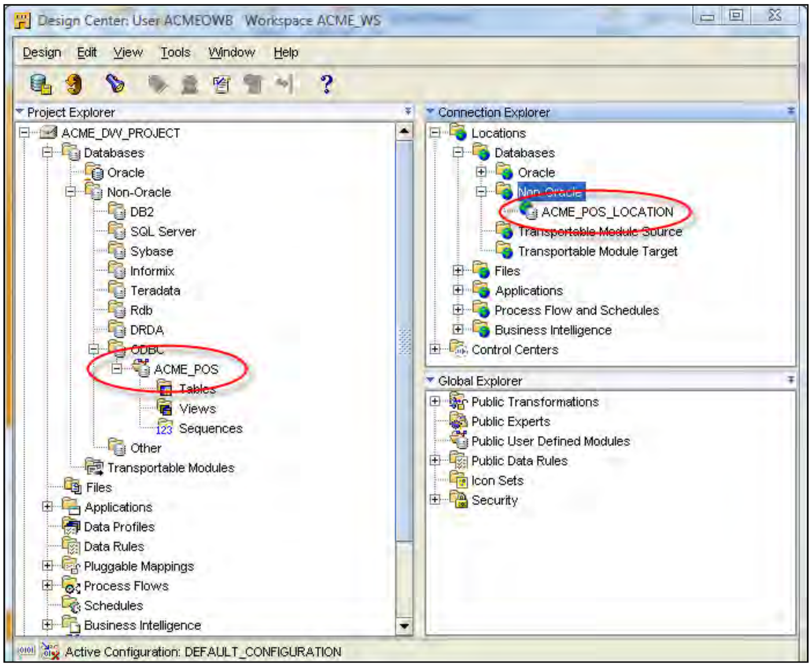
12. The **Import after finish** checkbox will be checked by default. But we're going to uncheck it, not because we're going to define any more modules, but because there is a problem that has not been fixed in this release of the database (11.1.0.6). This problem prevents us from being able to successfully import from a non-Oracle database and can cause an error when trying to test our connection. So, uncheck the box and click on the **Finish** button.

Download at Boykma.Com

There is a known bug in the ODBC gateway module code that heterogeneous services use to communicate with non-Oracle systems using ODBC. This bug should be fixed in the next release of the database as it can cause the location **Connection Test** feature and the metadata import feature to fail. The ability to retrieve data from a non-Oracle database is not affected by the bug and so we'll be able to load our data warehouse from the external database; but for now, we will just have to work around the bug.

Currently, release 11.1.0.7 is the most recent version of the database that is made available to customers of Oracle. The bug has not been fixed yet as of this release, but is scheduled for the next one. The download version from Oracle's web site is still 11.1.0.6, which we are using. The 11.1.0.7 update is available as of this writing only from Oracle's official support site, which is restricted to customers with fully licensed copies of the database. If you have a licensed version of the database, download and install 11.1.0.7 as it contains fixes for some other gateway bugs.

We are now back at the main Warehouse Builder interface and we can see that it has added our new module (**ACME_POS**) under **Databases | Non-Oracle** in the **Project Explorer**. In the **Connection Explorer**, if we expand the **Locations | Databases | Non-Oracle | ODBC** node or module, we'll see **ACME_POS_LOCATION** listed, which is our location that we just defined as part of the process of creating the module. This is shown in the following screenshot:



Download at Boykma.Com

Even if we had an error during the previous process of creating this connection, either caused by using the wrong name for the ODBC module or because of the bug just mentioned, we would still see these entries created. If we could fix whatever caused the error, we'd have a valid working connection without having to go back through the wizard to create it again.

Importing source metadata from a database

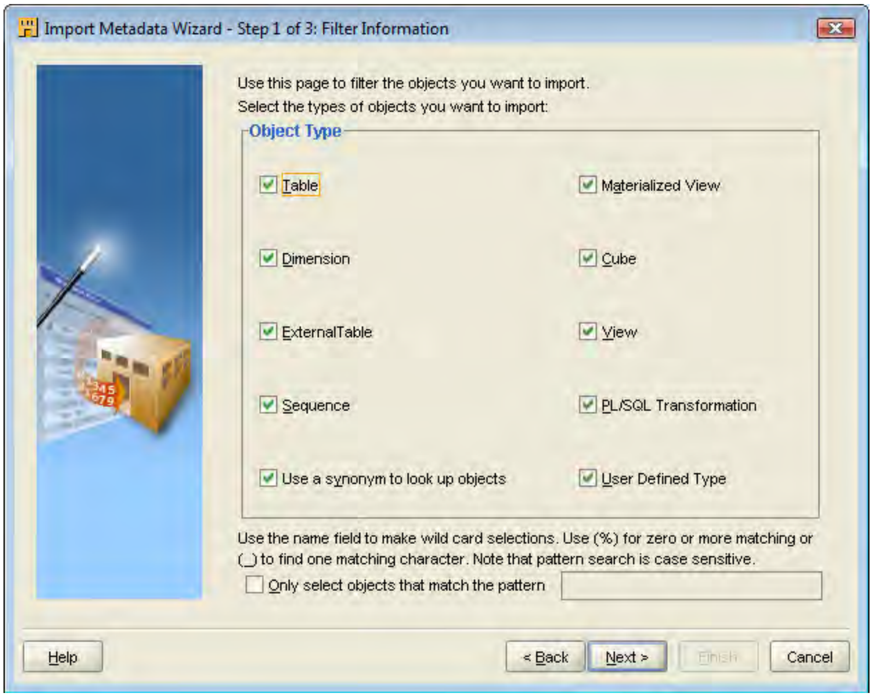
We are now at the point where we can finally import our metadata. However, we will not be able to import the metadata for our SQL Server database because of the bug mentioned above. We will have to manually define it. Importing from an Oracle database is not affected by the bug as we don't have to use ODBC to access an Oracle database. We'll walk through the process of importing from an Oracle database, which is very similar to the process in a non-Oracle database; so it will be a good exercise to walk through. After that, we will define the metadata for our SQL Server database using the **Data Object Editor**, which is the Warehouse Builder application for working with data objects. We are going to walk through the process for one source table here, the `Regions` table, and will leave the rest as an exercise for the reader to be done in a similar manner.

1. We are going to begin by right-clicking on the **ACME_WS_ORDERS** module name under the **Databases | Oracle** node in the **Project Explorer** and selecting **Import...** from the pop-up menu.

   We will then be presented with the **Import Metadata Wizard**. This is the same wizard that will be used for importing from any of the available source data options, databases, files, and so on. It will tailor its prompts for the particular type of source we selected.

2. Click on the **Next** button on the **Welcome** screen and we'll be presented with a screen labeled **Step 1** of the process where we choose what to import.

Download at Boykma.Com


Import Metadata Wizard - Step 1 of 3: Filter Information

Use this page to filter the objects you want to import.
Select the types of objects you want to import:

Object Type

- [✓] Table
- [✓] Materialized View
- [✓] Dimension
- [✓] Cube
- [✓] ExternalTable
- [✓] View
- [✓] Sequence
- [✓] PL/SQL Transformation
- [✓] Use a synonym to look up objects
- [✓] User Defined Type

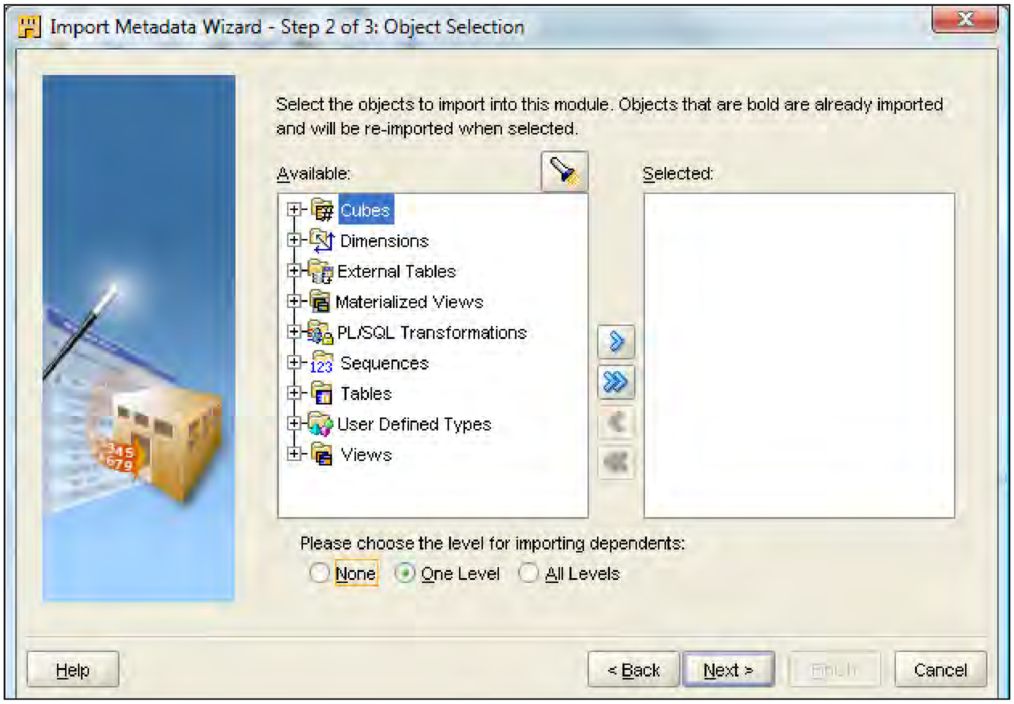
Use the name field to make wild card selections. Use (%) for zero or more matching or (_) to find one matching character. Note that pattern search is case sensitive.

- [ ] Only select objects that match the pattern

We can make selections on this screen to *filter out just what we want to import*, or we can leave everything checked to be able to import anything. This screen will appear slightly different depending on what type of source we're importing from. We will have all these options for an Oracle database, but for our ODBC connection to the SQL Server database, it will have checkboxes for just **Table** and **View**. There will also be a checkbox for whether to use synonyms to look up the objects, and a text box where we can enter a search pattern to use if we want to further refine what is available to us. We're going to leave everything checked, which is the default. Leaving the **Use a Synonyms** box checked means that if there are any synonyms defined (alternative names for database objects), then the import wizard will use those names and present them to us; otherwise it uses the underlying actual object names. As there are no synonyms being used in the **ACME_WS_ORDERS** source database, it will not make a difference whether it's checked or not.

Download at Boykma.Com

3. Click on **Next** to move on to **Step 2**:



Import Metadata Wizard - Step 2 of 3: Object Selection

Select the objects to import into this module. Objects that are bold are already imported and will be re-imported when selected.

Available:

- Cubes
- Dimensions
- External Tables
- Materialized Views
- PL/SQL Transformations
- Sequences
- Tables
- User Defined Types
- Views

Selected:

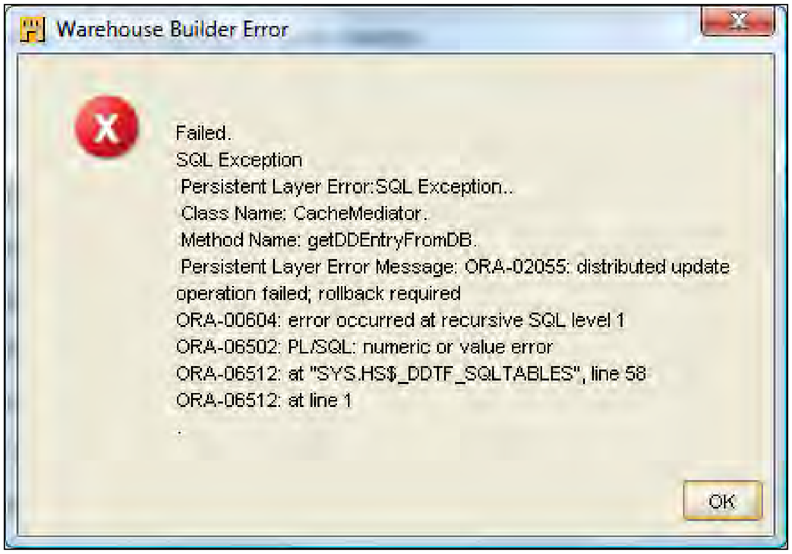
Please choose the level for importing dependents:

None [✓ One Level] All Levels

This screen is where we will choose the *specific objects that we wish to import*. There will be an entry in the left window for each of the boxes we left checked in **Step 1**. Notice (at the bottom) the buttons for choosing the level for importing dependents. The **Import Wizard** can automatically import other objects that might depend on the object we're selecting based on foreign key definitions that it detects in the source database. The number of levels means how far it goes in tracing foreign key relationships. If we say one level, which is the default, then it will import any tables that have foreign key relationships to the table selected but will not check those tables for relationships. If you say **All Levels**, then it will follow relationships until it doesn't find any further relationships. We're going to select all the tables in our **ACME_WS_ORDERS** schema to import, so this setting will not have an effect on what gets imported. Therefore, we'll leave it set to the default.

Download at Boykma.Com

4. Click on the plus sign beside the **Tables** entry to see the complete list of tables to choose from. We will see all of the web site orders' database tables that we discussed earlier. Clicking on the plus sign to expand an entry on this page is the first time the wizard will actually make a connection to the source database. If we saved the connection information before and didn't test it to make sure it worked, this is where we'll find out. In fact, if we were trying to import from a SQL Server database using the ODBC gateway, we would get the following error message when we try to expand the tables or views due to the known bug mentioned earlier:

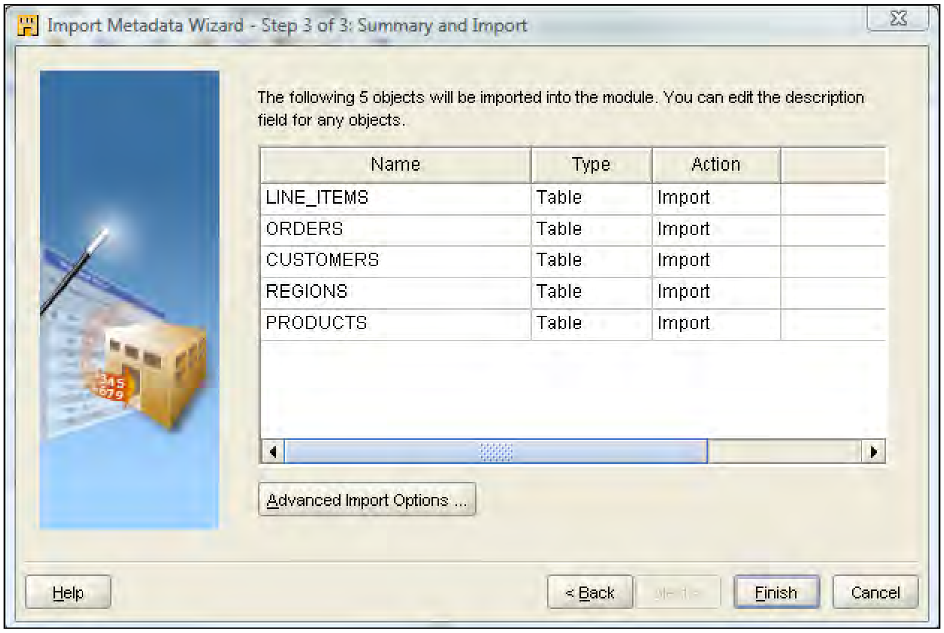


```
Failed.
SQL Exception
 Persistent Layer Error:SQL Exception..
 Class Name: CacheMediator.
 Method Name: getDDEntryFromDB.
 Persistent Layer Error Message: ORA-02055: distributed update
operation failed; rollback required
ORA-00604: error occurred at recursive SQL level 1
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at "SYS.HS$_DDTF_SQLTABLES", line 58
ORA-06512: at line 1
.
```

We are importing from an Oracle database now, so we will not see the above error. The table names will display under the tables entry. If we've already imported any of the tables previously, those will be displayed in bold. We can re-import them to pick up any changes that might have been made to them. We're going to click on all the tables, and then click on the single right arrow (**>**) in the middle of the screen to move those tables over to the right side. This will signify that we want to import them. As we want all the tables this time, we could alternatively click on the **Tables** entry itself and then on the single right arrow (**>**) to move that entry and everything in it over to the right. At this point, if we had one of the options checked for importing dependents and had not already selected the dependents to import, it would display a dialog box. This dialog box would inform us of any additional objects it detected as dependents that it was going to automatically add for us.

Download at Boykma.Com

5. We'll click on the **Next** button to proceed to the **Summary and Import** page where it will summarize the selections we've made and tell us the action it is going to take for each selection—whether to create or re-import the object. There is also an **Advanced Import Options...** button that will be available on that screen as we can see in the following image:


Import Metadata Wizard - Step 3 of 3: Summary and Import

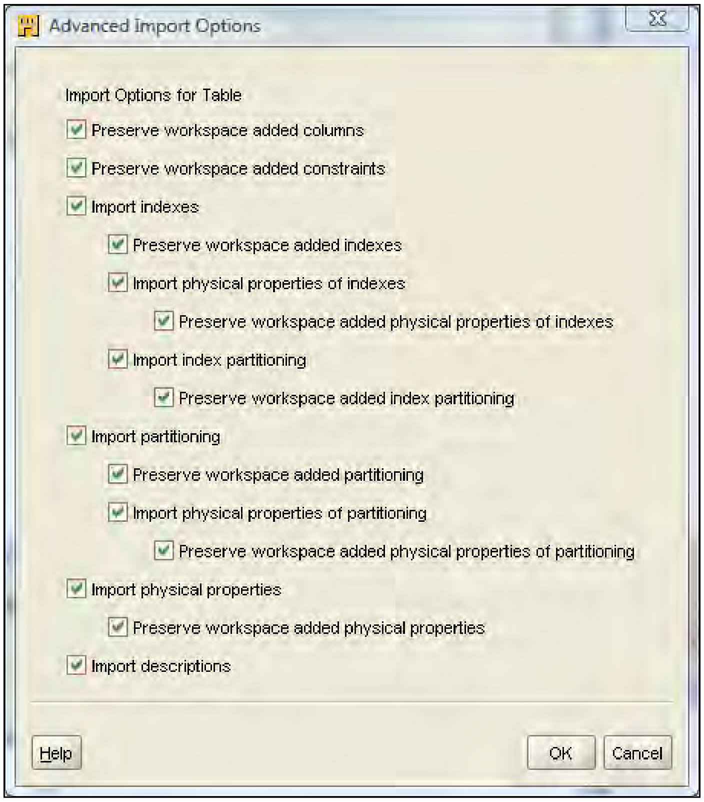
The following 5 objects will be imported into the module. You can edit the description field for any objects.

Name	Type	Action	
LINE_ITEMS	Table	Import	
ORDERS	Table	Import	
CUSTOMERS	Table	Import	
REGIONS	Table	Import	
PRODUCTS	Table	Import	

Advanced Import Options ...
Help
< Back
Next >
Finish
Cancel

Clicking on the **Advanced Import Options...** button presents us with a dialog box similar to the following screenshot:

Download at Boykma.Com



Advanced Import Options

Import Options for Table

- [✓] Preserve workspace added columns
- [✓] Preserve workspace added constraints
- [✓] Import indexes
  - [✓] Preserve workspace added indexes
  - [✓] Import physical properties of indexes
    - [✓] Preserve workspace added physical properties of indexes
  - [✓] Import index partitioning
    - [✓] Preserve workspace added index partitioning
- [✓] Import partitioning
  - [✓] Preserve workspace added partitioning
  - [✓] Import physical properties of partitioning
    - [✓] Preserve workspace added physical properties of partitioning
- [✓] Import physical properties
  - [✓] Preserve workspace added physical properties
- [✓] Import descriptions

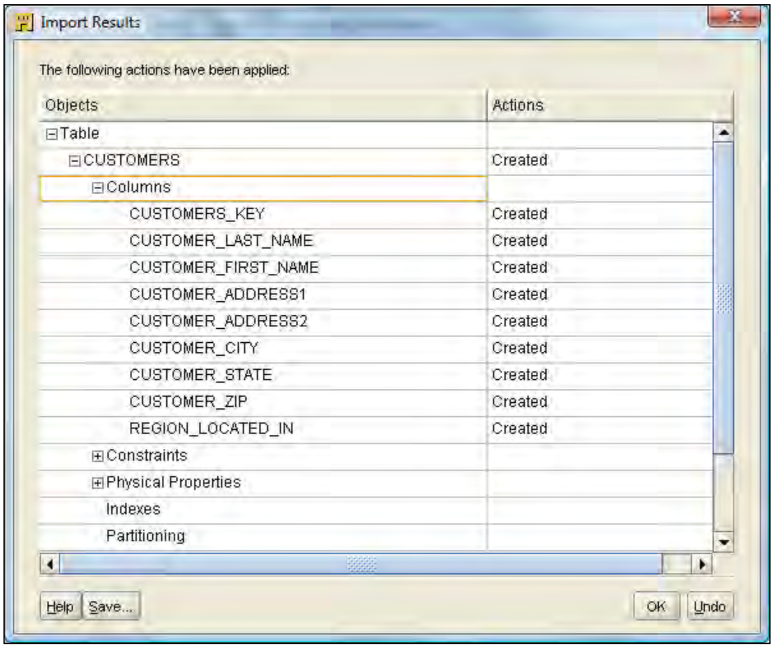
This dialog box will be slightly different, depending on the type of object and the type of source being imported. The screenshot we just saw is an example for a table from an Oracle database. It specifies whether to import certain features, such as indexes or physical properties, and also whether any possible changes we've made to the objects in the Warehouse Builder workspace after import should be preserved.

> Preserving changes made in the workspace would definitely be something we will need to consider when the ODBC Gateway bug is fixed and we're able to import objects from SQL Server again. We could go back and import all the table objects just to make sure they are up-to-date. But if we don't uncheck the boxes for preserving workspace changes and we've already created all the tables manually as we're about to do, it will leave all our column definitions in place and add new columns for each of the columns in the table. In that case, we would *definitely want to uncheck the preserve checkboxes* so our manual edits are replaced with a clean copy.

Download at Boykma.Com

We have verified on the **Summary and Import** screen that we have included everything we want to import and we don't need to bother with unchecking any of the advanced import options. So we will click on the **Finish** button, which will begin the import process.

6. During the import, a status dialog box will be displayed showing the progress of the import as each object is imported. When it completes, we'll be presented with the final **Import Results** screen showing the status of the import. We can click on the plus sign beside each entry to see the details as shown in the following screenshot with the `Customers` table expanded to show each of the columns:



The following actions have been applied:

Objects	Actions
⊟Table	
⊟CUSTOMERS	Created
⊟Columns	
CUSTOMERS_KEY	Created
CUSTOMER_LAST_NAME	Created
CUSTOMER_FIRST_NAME	Created
CUSTOMER_ADDRESS1	Created
CUSTOMER_ADDRESS2	Created
CUSTOMER_CITY	Created
CUSTOMER_STATE	Created
CUSTOMER_ZIP	Created
REGION_LOCATED_IN	Created
⊞Constraints	
⊞Physical Properties	
Indexes	
Partitioning	

The other buttons you have on this screen are:

- A **Save** button which will allow us to save an **MDL** file of the activity we just accomplished. An MDL is a file the Warehouse Builder can save that contains information from the model that can be imported later. We can use it to transfer model information between databases if we have more than one, or we can use it as a backup.

Download at Boykma.Com

    ◦ An **Undo** button that we could click on at this point to cancel the import. The **Import Metadata Wizard** has not actually saved any information to the database yet, so clicking on the **Undo** button will just throw away what we've done so far and not make any changes to the database.
    ◦ The **OK** button will save the changes to the module in **Project Explorer** from which we performed the import.

  In this case, clicking on **OK** is going to save the imported tables in the **ACME_WS_ORDERS** module that we created under the **Databases | Oracle** node. We can verify this by going back to the **Project Explorer** window and expanding that module if it's not already expanded by clicking on the plus sign, and we should see the list of tables.

Congratulations! We've imported our first set of objects into the Warehouse Builder. When the ODBC gateway bug is fixed in the database and we can once again import successfully using it, the import process is identical. The only difference will be in a couple of screens and options available. The following is a list of some of the key differences for reference:

- The screen labeled **Step 1** that is used for choosing objects to import will be restricted to only tables and views
- Advanced import options will have far fewer options available; mainly preserve options for changes made in the workspace

Even though we're not able to import from a SQL Server database using the ODBC gateway, we still need to define these tables in our project so that we can pull data from them. Let's take a look at the Data Object Editor now for manually defining our SQL Server tables.

We should save our work at this point. So we'll select **Design | Save All** from the toolbar menu of the **Design Center** application or press the *Ctrl* + *S* key combination to save our work.

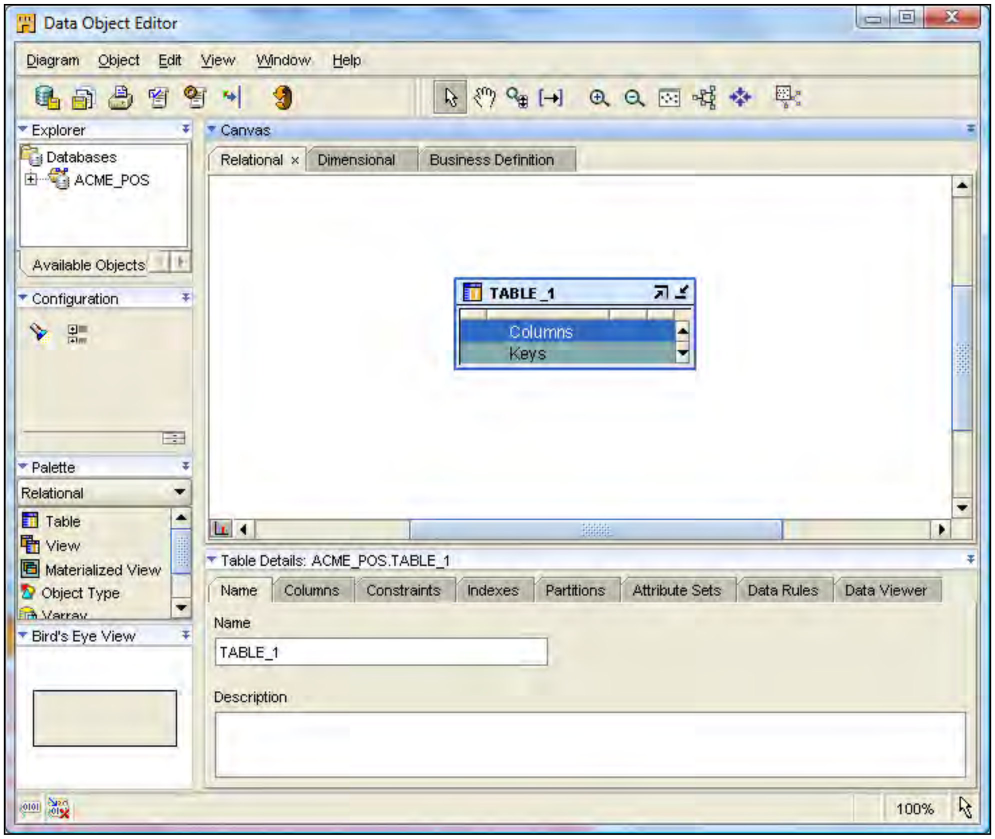
Download at Boykma.Com

Defining source metadata manually with the Data Object Editor

Before we can continue building our data warehouse, we must have all our source table metadata created. As we were unable to import our source metadata automatically for the SQL Server database for the POS transactional database, we must create the source metadata manually. It is not a particularly difficult task. However, attention to detail is important to make sure what we manually define in the Warehouse Builder actually matches the source tables we're defining. The tool the Warehouse Builder provides for creating source metadata is the Data Object Editor, which is the tool we can use to create any object in the Warehouse Builder that holds data such as database tables. The steps to manually define the source metadata using Data Object Editor are:

1. To start building our source tables for the POS transactional SQL Server database, let's launch the OWB Design Center if it's not already running. Expand the **ACME_DW_PROJECT** node and take a look at where we're going to create these new tables. We have imported the source metadata into the SQL Server ODBC module so that is where we will create the tables. Navigate to the **Databases | Non-Oracle | ODBC** node, and then select the **ACME_POS** module under this node. We will create our source tables under the **Tables** node, so let's right-click on this node and select **New,** from the pop-up menu. As no wizard is available for creating a table, we are using the Data Object Editor to do this.
2. Upon selecting **New**, we are presented with the **Data Object Editor** screen. It's a clean slate that we get to fill in, and will look similar to the following screenshot:

Download at Boykma.Com



There are a number of facets to this interface but we will cover just what we need now in order to create our source tables. Later on, we'll get a chance to explore some of the other aspects of this interface for viewing and editing a data object. The fields to be edited in this **Data Object Editor** are as follows:

- The first tab it presents to us is the **Name** tab where we'll give a name to the first table we're creating. We should not make up table names here, but use the actual name of the table in the SQL Server database. Let's starts with the `Items` table. We'll just enter its name into the **Name** field replacing the default, **TABLE_1**, which it suggested for us. The Warehouse Builder will automatically capitalize everything we enter for consistency, so there is no need to worry about whether we type it in uppercase or lowercase.

Download at Boykma.Com

- ° Let's click on the **Columns** tab next and enter the information that describes the columns of the `Items` table. How do we know what to fill in here? Well, that is easy because the names must all match the existing names as found in the source POS transactional SQL Server database. For sizes and types, we just have to match the SQL Server types that each field is defined as, making allowances for slight differences between SQL Server data types and the corresponding Oracle data types.

The following will be the columns, types, and sizes we'll use for the `Items` table based on what we found in the `Items` source table in the POS. transaction database:

```
ITEMS_KEY number(22)
ITEM_NAME varchar2(50)
ITEM_CATEGORY varchar2(50)
ITEM_VENDOR number(22)
ITEM_SKU varchar2(50)
ITEM_BRAND varchar2(50)
ITEM_LIST_PRICE number(6,2)
ITEM_DEPT varchar2(50)
```

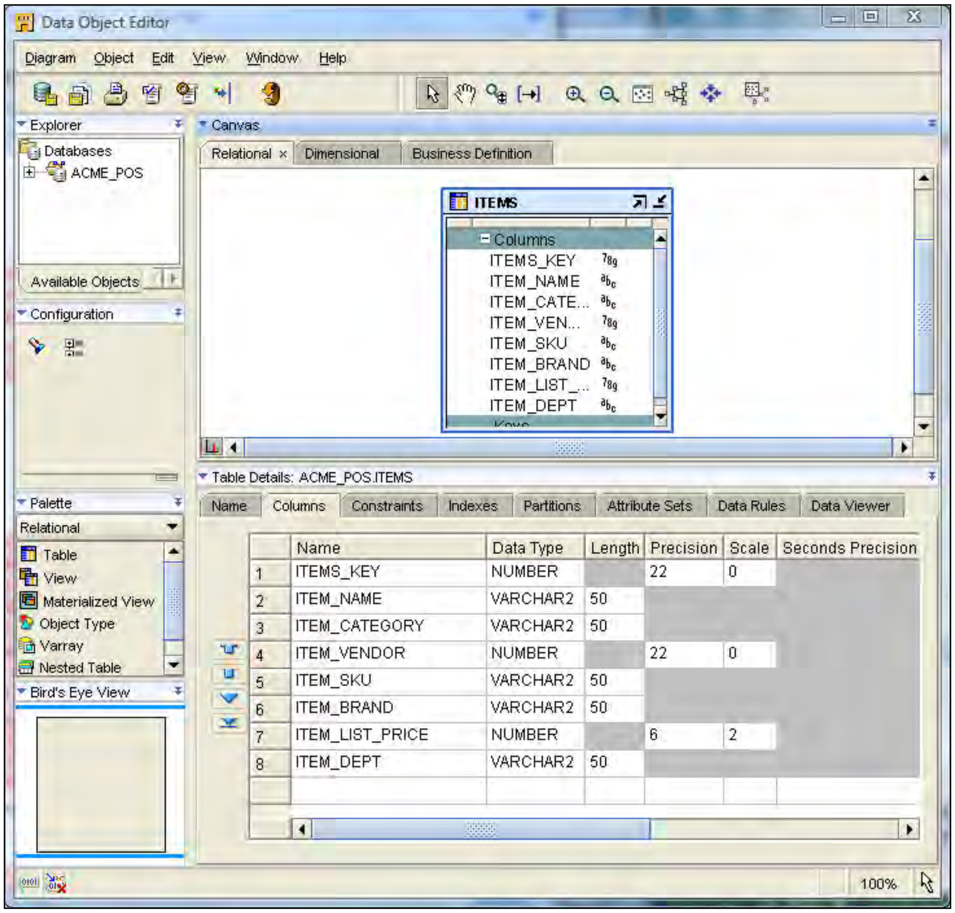
We'll enter each of these column names on the **Columns** tab of the **Data Object Editor** for the `Items` table; and as we enter each name, it will suggest data types and sizes, which may or may not be adequate. It makes a best guess based on what we enter for the name, and may or may not relate to the source data type and size. For `ITEMS_KEY`, it suggests a `number` with precision `22`. For `ITEM_VENDOR`, it actually suggested a `varchar2` type. We simply change it to match the `ITEMS_KEY`, as we see in the SQL Server database that both these fields are defined as type `INT` and `22` for the precision is large enough to hold an integer from SQL Server. An integer is a four-byte number, no larger than 2,147,483,647. The other character fields in the SQL Server `Items` table are all of the `varchar` type, which is the SQL Server equivalent to a `varchar2` in Oracle. So we make sure they are `varchar2` with sizes that match. The `ITEM_LIST_PRICE` is defined with both a precision and scale because that is a decimal number in the `Items` table in SQL Server with that precision and scale.

Download at Boykma.Com

Precision and scale of numbers

Properties of number data types can include a precision and scale. Oracle allows a `number` data type to be specified without indicating a specific precision and scale. Precision indicates the maximum number of digits the number can contain, and scale indicates the number of decimal places to the right of the decimal. If we don't specify them, Oracle Database will accept a number of any precision and scale as long as the number doesn't fall outside the range allowed, which is between 1.0 x 10^{-130} and 1.0 x 10^{126}. We would specify a precision and scale to enforce greater data integrity in the database. For example, if we enter a number that has more digits than the specified precision, it will generate an error even though the number might still fall in the acceptable range.

When completed, our column list should look like the following screenshot:



	Name	Data Type	Length	Precision	Scale	Seconds Precision
1	ITEMS_KEY	NUMBER		22	0	
2	ITEM_NAME	VARCHAR2	50			
3	ITEM_CATEGORY	VARCHAR2	50			
4	ITEM_VENDOR	NUMBER		22	0	
5	ITEM_SKU	VARCHAR2	50			
6	ITEM_BRAND	VARCHAR2	50			
7	ITEM_LIST_PRICE	NUMBER		6	2	
8	ITEM_DEPT	VARCHAR2	50			

Download at Boykma.Com

We don't have to worry about specifying information for any of the other tabs for this source data. The important details are the column names, and their types and sizes. Later in the book, we'll have a chance to revisit the Data Object Editor and discuss the remainder of the tabs.

3. We can save our work at this point and close the **Data Object Editor** window now before proceeding. So we'll select **Diagram | Save All** from the toolbar menu of the **Data Object Editor**, or press the *Ctrl* + *S* key combination to save our work. We exit from the Data Object Editor by selecting **Diagram | Close Window** from **Data Object Editor**.

We now have to continue this process to define the metadata for the remaining SQL Server tables that we'll need. The process is identical; just change the names of the tables and the types and sizes of the columns to match their respective tables. We will not need all the tables defined in the `ACME_POS` database—only a subset is used throughout the remainder of the book to build the data warehouse. The tables needed are the `POS_TRANSACTIONS`, `REGISTERS`, `STORES`, and `REGIONS` tables. The column information for each of them is provided here for reference and help in creating the corresponding tables in the Warehouse Builder:

`POS_TRANSACTIONS`

```
POS_TRANS_KEY number(22)
SALES_QUANTITY number(22)
SALES_ASSOCIATE number(22)
REGISTER number(22)
ITEM_SOLD number(22)
DATE_SOLD date
AMOUNT number(10,2)
```

`REGISTERS`

```
REGISTERS_KEY number(22)
REGISTER_MANUFACTURER varchar2(60)
MODEL varchar2(50)
LOCATION number(22)
SERIAL_NO varchar2(50)
```

`STORES`

```
STORES_KEY number(22)
STORE_NAME varchar2(50)
STORE_ADDRESS1 varchar2(60)
STORE_ADDRESS2 varchar2(60)
STORE_CITY varchar2(50)
STORE_STATE varchar2(50)
STORE_ZIP varchar2(50)
REGION_LOCATED_IN number(22)
STORE_NUMBER varchar2(10)
```

Download at Boykma.Com

```
REGIONS
REGIONS_KEY number(22)
REGION_NAME varchar2(50)
CONTINENT varchar2(50)
COUNTRY varchar2(50)
```

Case sensitivity of column names in SQL Server

We used all uppercase for the column names above because the Warehouse Builder defaults the case to upper for any column name that we enter. However, this could cause a problem later while retrieving data from the SQL Server tables using those column names if the case does not match the way they are defined in SQL Server. SQL Server will allow mixed case for column names, but the Oracle database assumes all uppercase for column names. It is possible to query a mixed-case column name in SQL Server from an Oracle database, but the name must be enclosed in double quotes. The code generated by the Warehouse Builder recognizes this and puts double quotes around any references to column name. If the import had worked, it would have created the column names with matching case and there would have been no problem. However, when the Data Object Editor manually enters columns, it has no option to enter a mixed-case name. We'll run into errors later if the corresponding SQL Server column names are not in all uppercase. The database scripts that can be downloaded from the Packt web site (`http://www.packtpub.com/files/code/5746_Code`) to build the database contain the column names in all uppercase to avoid any problem.

Be sure to save each table as it is created to make sure no work gets lost. When all the tables are entered, our defining and importing of source metadata is completed.

Importing source metadata from files

One final object type we need to discuss before we wrap up the source metadata importing and defining is the import of metadata from a file. The Warehouse Builder can take data from a flat file and make it available in the database as if it were a table, or just load it into an existing table. The metadata that describes this file must be defined or imported in the Warehouse Builder. The file format must be delimited, usually with commas separating each column and a carriage return at the end of a record (**CSV file**). The option to use a flat file greatly expands the flexibility of the Warehouse Builder because now it allows us to draw upon data from other sources, and not just databases. That can also be of great assistance even in loading data from a database if the database is not directly network accessible to our data warehouse. In that case, the data can be exported out of the source database tables and saved to a CSV file for us to import.

Download at Boykma.Com

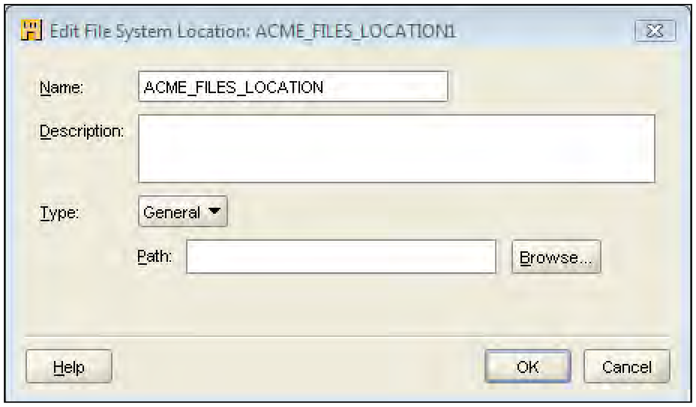
For our ACME Toys and Gizmos company data warehouse, we've been provided a flat file. This file contains information for counties in the USA that the management wanted to see in the data warehouse to allow analyzing by county. For stores in the USA, the store number includes a code that identifies the county the store is located in, and the flat file we've been provided with contains the cross reference of the code to the county that we'll need.

The file name is `counties.csv` and it is available in the download files from the Packt web site at `http://www.packtpub.com/files/code/5746_Code`.

The process of creating the module and importing the metadata for a flat file is still the same as importing from Oracle or non-Oracle databases, but there are some minor differences. The steps involved in creating the module and importing the metadata for a flat file are:

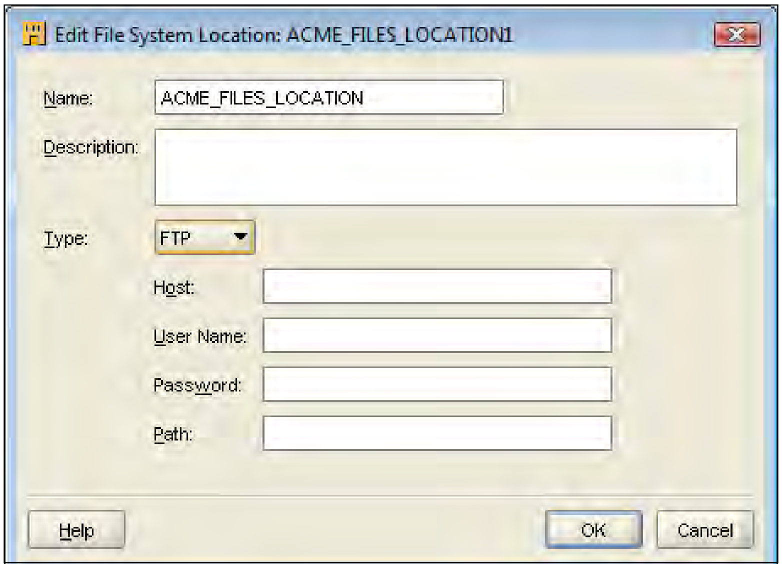
1. The first task we need to perform, as we did earlier for the source databases, is to create a new module to contain our file definition. If we look in the **Project Explorer** under our project, we'll see that there is a **Files** node right below the **Databases** node. We will launch the **Create Module Wizard** the same way as we did earlier, but we'll do it on the **Files** node and not the **Databases** node. We'll right-click on the **Files** node and select **New** from the pop-up menu to launch the wizard.
2. When we click on the **Next** button on the **Welcome** screen, we notice a slight difference already. The **Step 1** of the **Create Module** wizard only asks for a name and description. The other options we had for databases above are not applicable for file modules. We'll enter a name of **ACME_FILES** and click on the **Next** button to move to **Step 2**.
3. We need to edit the connection in **Step 2** just as we did for the database previously. So we'll click on the **Edit** button and immediately notice the other major difference in the **Create Module Wizard** for a file compared to a database. As we see in the following image, it only asks us for a name, a description, and the path to the folder where the files are.

Download at Boykma.Com



Name: ACME_FILES_LOCATION
Description:
Type: General
Path:

4. The **Name** field is prefilled with the suggested name based on the module name. As it did for the database module location names, it adds that number **1** to the end. So, we'll just edit it to remove the number and leave it set to **ACME_FILES_LOCATION**.
5. Notice the **Type** drop-down menu. It has two entries: **General** and **FTP**. If we select **FTP** (**File Transfer Protocol**—used for getting a file over the network), it will ask us for slightly more information as shown in the following image:



Name: ACME_FILES_LOCATION
Description:
Type: FTP
Host:
User Name:
Password:
Path:

The **FTP** option can be used if the file we need is located on another computer. We will need to know the name of the computer, and have a logon username and password to access that computer. We'll also need

Download at Boykma.Com

to know the path to the location of the file.

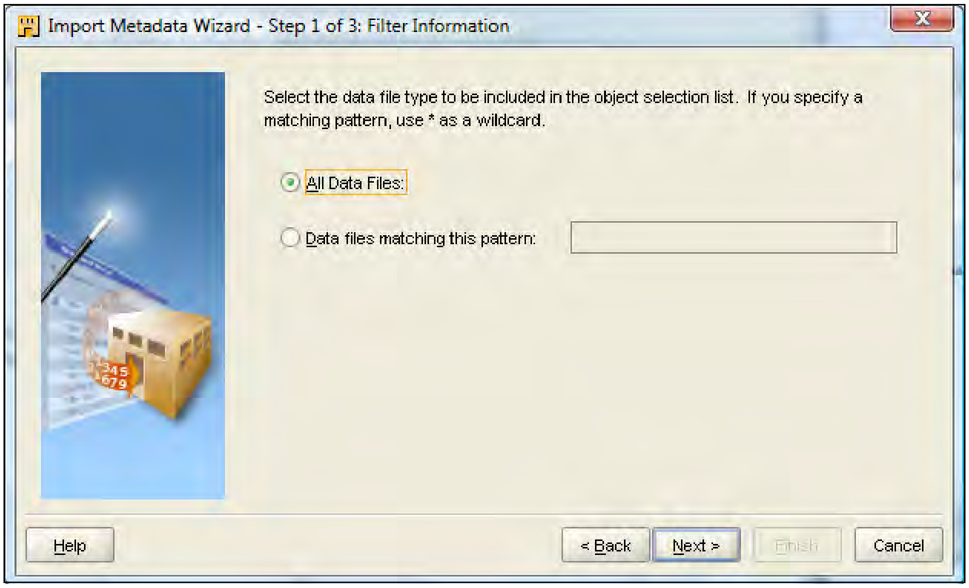
6. The simplest option is to store the file on the same computer on which we are running the database. This way, all we have to do is enter the path to the folder that contains the file. We should have a standard path we can use for any files we might need to import in the future. So we create a folder called `GettingStartedWithOWB_files`, which we'll put in the `D:` drive. Choose any available drive with enough space and just substitute the appropriate drive letter. We'll click on the **Browse** button on the **Edit File System Location** dialog box, choose the **D:\GettingStartedWithOWB_files** path, and click on the **OK** button.
7. We'll then check the box for **Import after finish** and click on the **Finish** button.

That's it for the **Create Module Wizard** for files. It's really very straightforward.

The **Import Metadata Wizard** appears next and will have some additional differences, which we'll cover now. We'll work through this one in a little more detail as it is different from importing from a database. If the **Import Wizard** has not appeared, then just right-click on the module name under the **Files** node under our project, and select **Import**. The following are the steps to be performed on the **Import Metadata Wizard**:

1. The first difference we're going to notice is on the very first screen for selecting **Filter Information**. As we're working with files in this case, and not a database, the only option available for filtering what we choose to import is the file name. There are two options, **All Data Files** and **Data files matching this pattern**, as shown in the following screenshot:

Download at Boykma.Com


Import Metadata Wizard - Step 1 of 3: Filter Information

Select the data file type to be included in the object selection list. If you specify a matching pattern, use * as a wildcard.

(●) All Data Files:

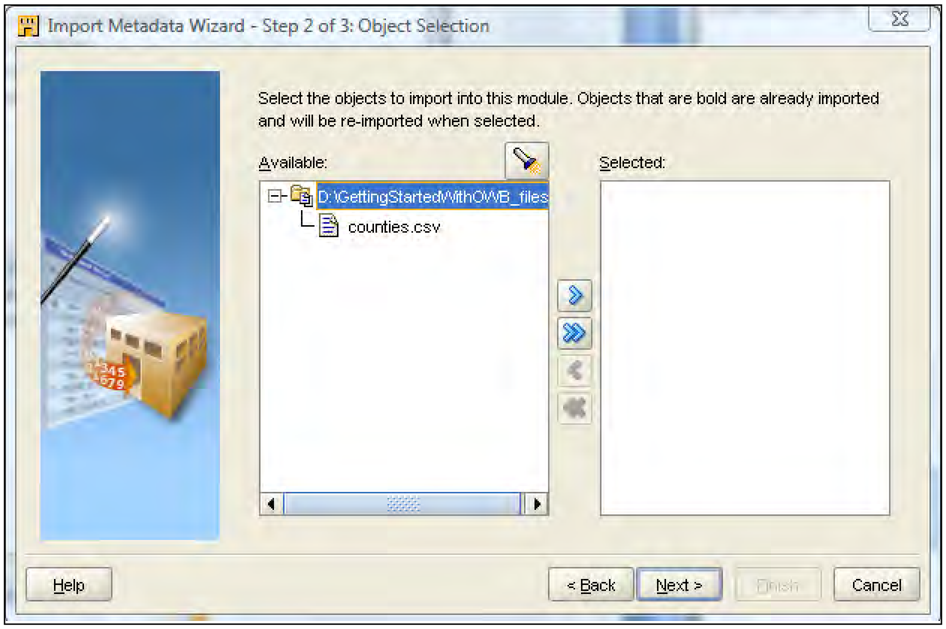
( ) Data files matching this pattern:

We're going to leave this set to the default for all files and just choose the file we want in the next step.

2. The next screen is the **Object Selection** screen, which is similar to the one we used for selecting the database objects to import from. But this one shows the folder contents of the files matching the filter criteria we specified on the previous screen. If we click on the plus sign beside the folder name, we'll see a list of files that shows the single `counties.csv` file as that's the only file in that folder, as shown in the following screenshot.

Download at Boykma.Com

If no file appears, it means we forgot to copy the file into that folder. We can try again by copying it now, clicking on the **Back** button, and then on the **Next** button to get back to this screen:



Import Metadata Wizard - Step 2 of 3: Object Selection

Select the objects to import into this module. Objects that are bold are already imported and will be re-imported when selected.

Available:

D:\GettingStartedWithOWB_files

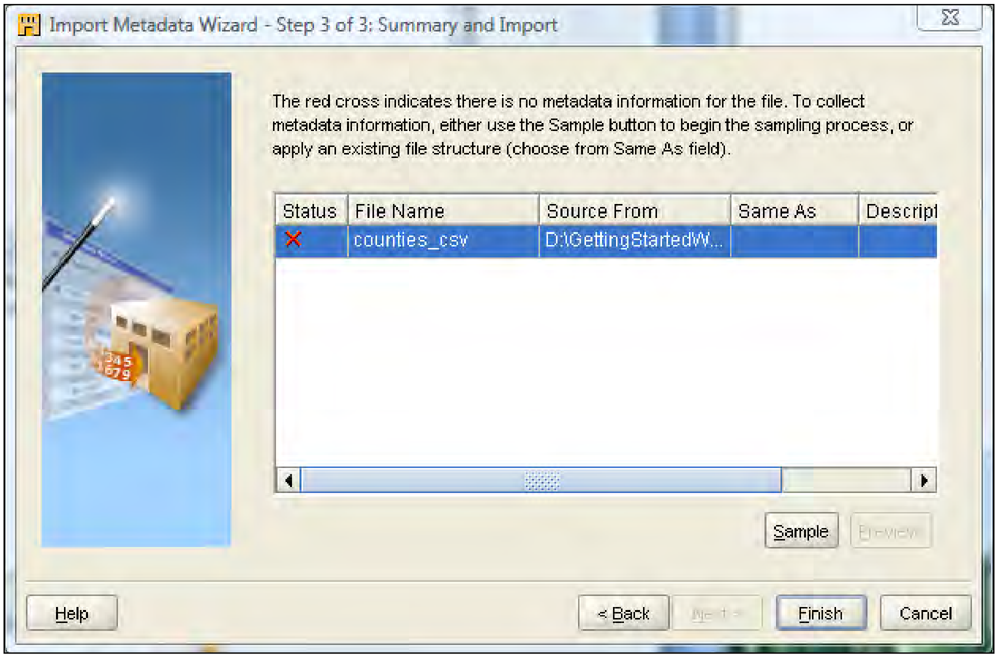
counties.csv

Selected:

We'll also notice that there is no option for specifying the dependents in this case. The **Import Metadata Wizard** has no way to determine what files depend on each other as it can for database tables. The **Wizard** can look up that information in the database, but there is no way to look it up in a file. So if we had more than one file to import, we'd have to make sure we selected all the files that might depend in some way on this one.

3. We select the `counties.csv` file, click on the right arrow (**>**), and then move on to the **Next** step. This brings us to the next screen, which looks like the following:

Download at Boykma.Com


Import Metadata Wizard - Step 3 of 3: Summary and Import

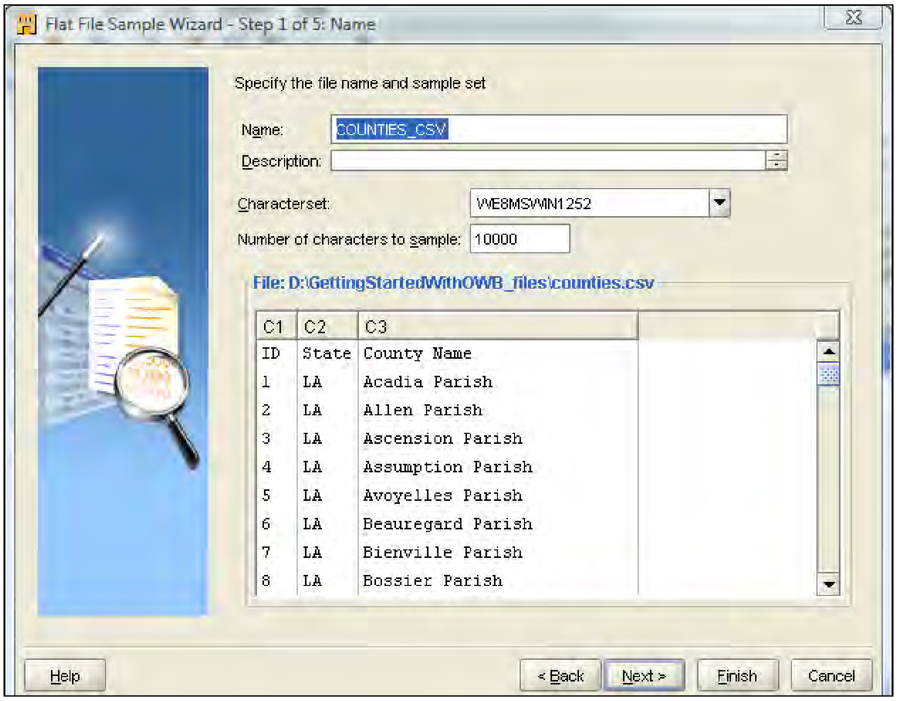
The red cross indicates there is no metadata information for the file. To collect metadata information, either use the Sample button to begin the sampling process, or apply an existing file structure (choose from Same As field).

Status	File Name	Source From	Same As	Descript
✗	counties_csv	D:\GettingStartedW...		

4. This looks rather different from the **Summary and Import** page of the database imports, doesn't it? One main difference is the red **X** in the **Status** column. This is a file we're importing and all it contains is a list of comma-separated values for each row. It contains no other information that could describe to the **Import Wizard** what these column names are called, or what kind of data the columns hold. Databases have system tables that store all the information that the **Wizard** can use to look up without having to ask us, but it needs us to specify it for files. The red **X** indicates that we have not yet specified that information for this file.

Download at Boykma.Com

5. We use the **Sample** button to enter that information. When we click on that button, we are presented with an entirely new screen that we haven't seen before. This is **Flat File Sample Wizard**, which has now been started. On the **Welcome** screen, we click on **Next** to move on to **Step 1** as shown in the following screenshot:



Specify the file name and sample set

Name: COUNTIES_CSV

Description:

Characterset: WE8MSWIN1252

Number of characters to sample: 10000

File: D:\GettingStartedWithOWB_files\counties.csv

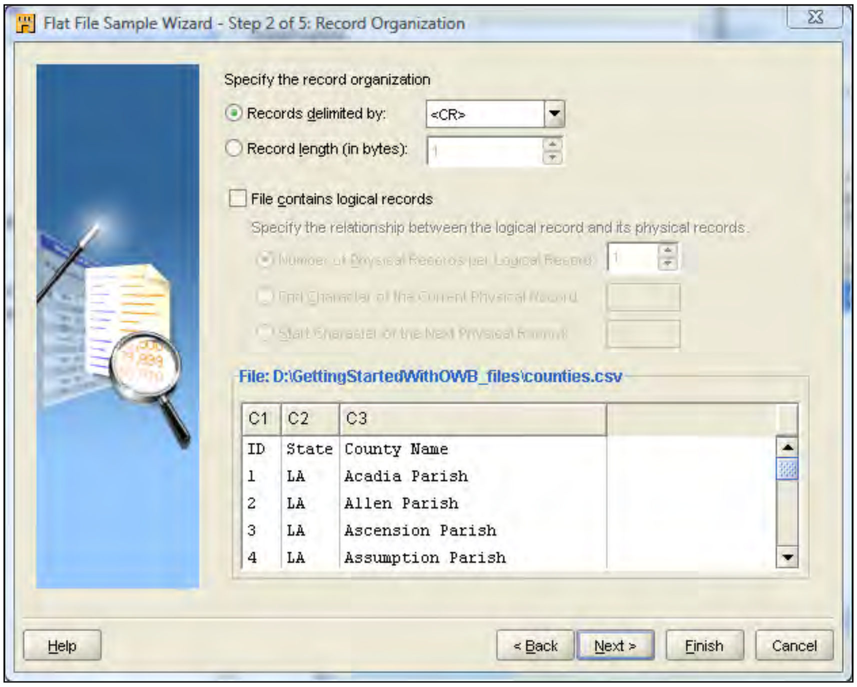
C1	C2	C3
ID	State	County Name
1	LA	Acadia Parish
2	LA	Allen Parish
3	LA	Ascension Parish
4	LA	Assumption Parish
5	LA	Avoyelles Parish
6	LA	Beauregard Parish
7	LA	Bienville Parish
8	LA	Bossier Parish

This screen displays the information the wizard pulled out of the file, displayed as columns of information. It knows what's in the columns because the file has each column separated by a comma, but doesn't know at this point what type of data or column name to use for each column—so it just displays the data. It picks a name based on the file name, which is fine. So we'll leave this and the remaining options set to the default. The following are the options on this screen:

- More information about what those fields mean can be found by clicking on the **Help** button.
- The **Characterset** is language related. For English language, the default character set will work fine.
- The **Number of characters to sample** determines how much of the file the wizard will read to get an idea of what's in it. If we were to import a file with multiple record types, this field might have come into play. But for our purposes, the default is enough.

Download at Boykma.Com

6. In the next step, we'll specify even more information about the characteristics of the file as shown in the following screenshot:


Flat File Sample Wizard - Step 2 of 5: Record Organization
Specify the record organization
Records delimited by: <CR>
Record length (in bytes): 1
File contains logical records
Specify the relationship between the logical record and its physical records
Number of Physical Records per Logical Record
End Character of the Current Physical Record
Start Character of the Next Physical Record
Help
< Back
Next >
Finish
Cancel

File: D:\GettingStartedWithOWB_files\counties.csv

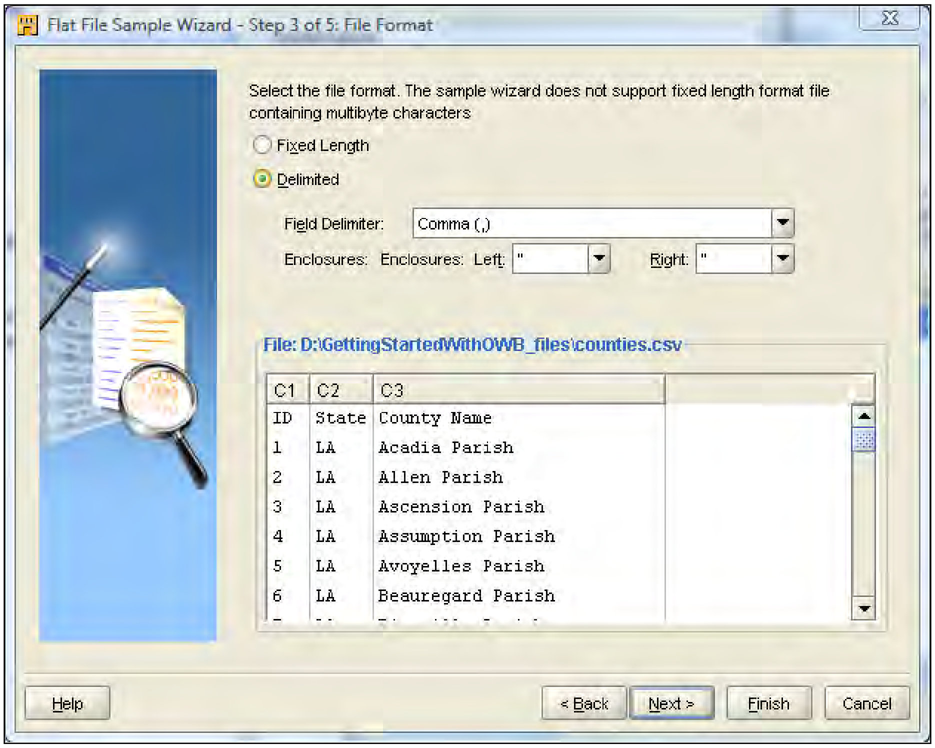
C1	C2	C3
ID	State	County Name
1	LA	Acadia Parish
2	LA	Allen Parish
3	LA	Ascension Parish
4	LA	Assumption Parish

This is where we'll specify the record information, so the wizard will know how to tell where one record ends and the next one begins. The commas only determine where one column ends and another begins. But where does the next row start? We can see by the display that it already seems to have figured that out because it assumes that a **carriage return <CR>** character will indicate the end of a row. This is an invisible character that gets entered into a text file when we press the *Enter* key while editing a file to move to the next line. It's possible that we might get a file with some other character indicating the end of a row, but our files use the carriage return, which is the most common. So we'll leave it set to that.

The other option here is to indicate whether or not the **File contains logical records**. Our file contains a physical record for each logical record. In other words, each row in the file is only one record. It's possible that one record's worth of information in a table might be contained in more than one physical row in the file. If this is the case, we can check that box and then specify the number of physical records that make up one complete logical record.

Download at Boykma.Com

7. The next step is where we specify the **File Format**, which tells the wizard how the columns are separated as shown here:



Select the file format. The sample wizard does not support fixed length format file containing multibyte characters

Fixed Length [✓ Delimited]

Field Delimiter: Comma (,)

Enclosures: Enclosures: Left: " Right: "

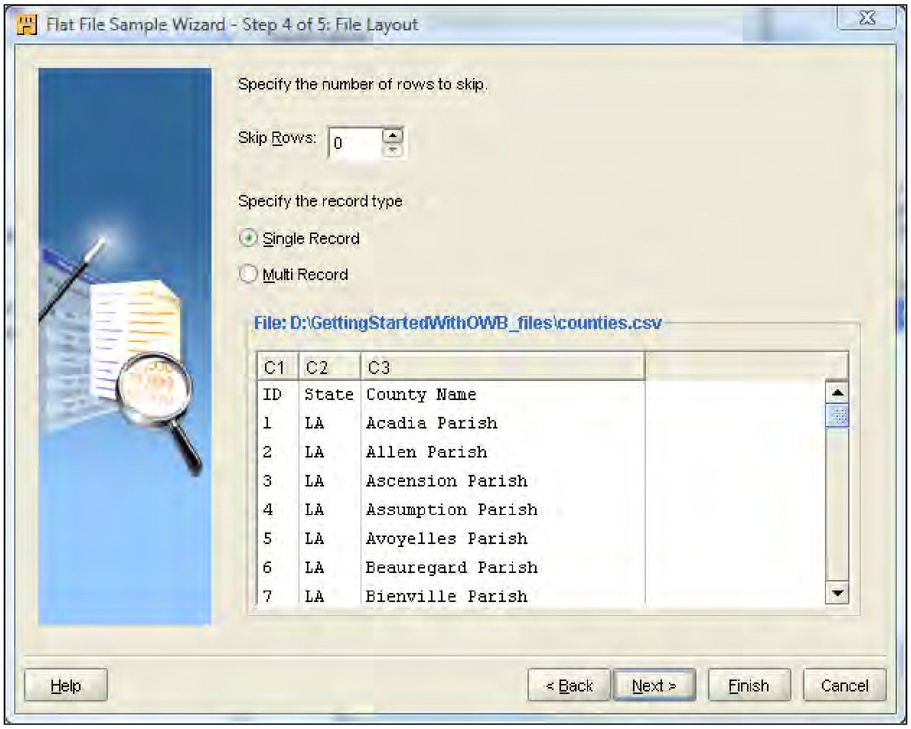
File: D:\GettingStartedWithOWB_files\counties.csv

```
C1  C2     C3
ID  State  County Name
1   LA     Acadia Parish
2   LA     Allen Parish
3   LA     Ascension Parish
4   LA     Assumption Parish
5   LA     Avoyelles Parish
6   LA     Beauregard Parish
```

Our records are separated by **Comma,** and that is the default, so we'll leave it at that. The **Enclosures:** selection is OWB's way of specifying the characters that surround the text values in the file. Frequently, in text-based files such as CSV files, the text is differentiated from numerical values by surrounding the text values in double quotes, single quotes, or something similar. This is where we specify, the characters if any, that surround the text-field values in this file. As our file does not use any character to surround text values and does not contain any double quotes, this setting will have no effect and we can safely ignore it. It is possible that double quotes, or any of the other characters available from the drop-down menu, might appear in text strings in the file but not as delimiters. We would need to set this to blank to indicate that there's no text delimiter in that case so that it wouldn't get confused.

Download at Boykma.Com

8. Moving to the next step, we get to specify the number of rows to skip and whether the file contains more than one record type as shown here:



Specify the number of rows to skip.

Skip Rows: 0

Specify the record type

(●) Single Record

( ) Multi Record

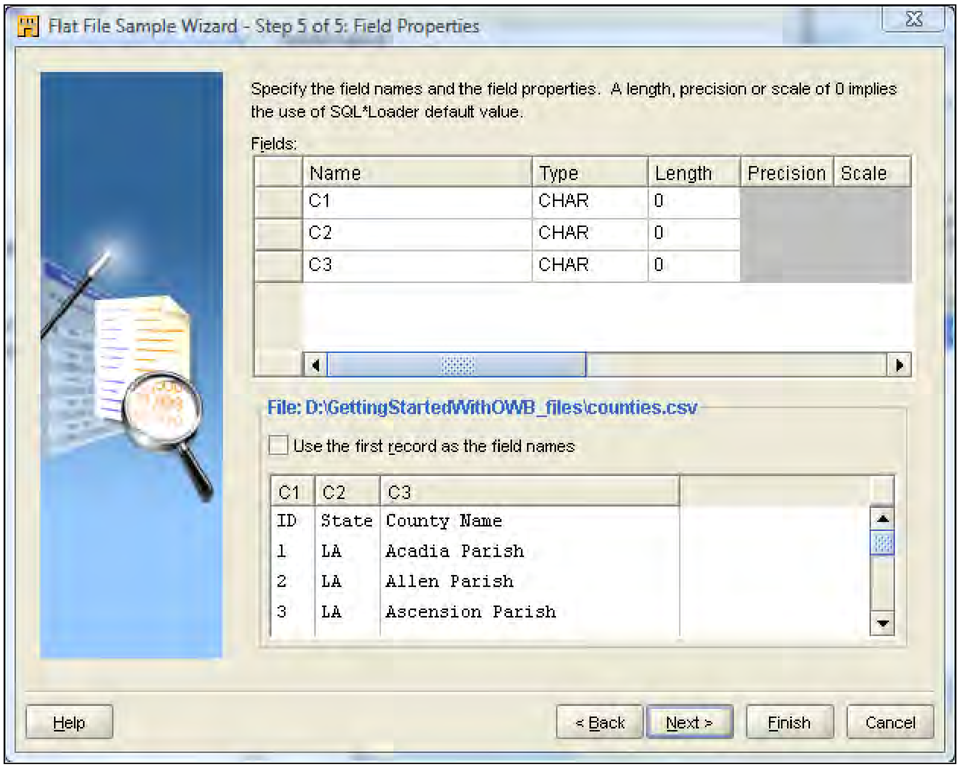
File: D:\GettingStartedWithOWB_files\counties.csv

C1	C2	C3
ID	State	County Name
1	LA	Acadia Parish
2	LA	Allen Parish
3	LA	Ascension Parish
4	LA	Assumption Parish
5	LA	Avoyelles Parish
6	LA	Beauregard Parish
7	LA	Bienville Parish

Sometimes the provided files may contain a number of rows of preliminary text information before the actual rows of data start. We can tell the wizard to skip over those rows at the start as they aren't formatted like the rows of data. All the rows in our file are formatted the same, so we will leave the default set to **0** as we want it to read all the rows. But wait, what about that first row? It looks like the header information that identifies what the row contains. Wouldn't we want it to skip over that so we're just looking at data? The answer is no, and we'll see why on the next screen.

Download at Boykma.Com

The records are all of a **Single Record** type. So leaving that checked as the default, we move on to the next step by clicking on the **Next** button as shown in the following screenshot:



Specify the field names and the field properties. A length, precision or scale of 0 implies the use of SQL*Loader default value.

Fields:

	Name	Type	Length	Precision	Scale
	C1	CHAR	0		
	C2	CHAR	0		
	C3	CHAR	0		

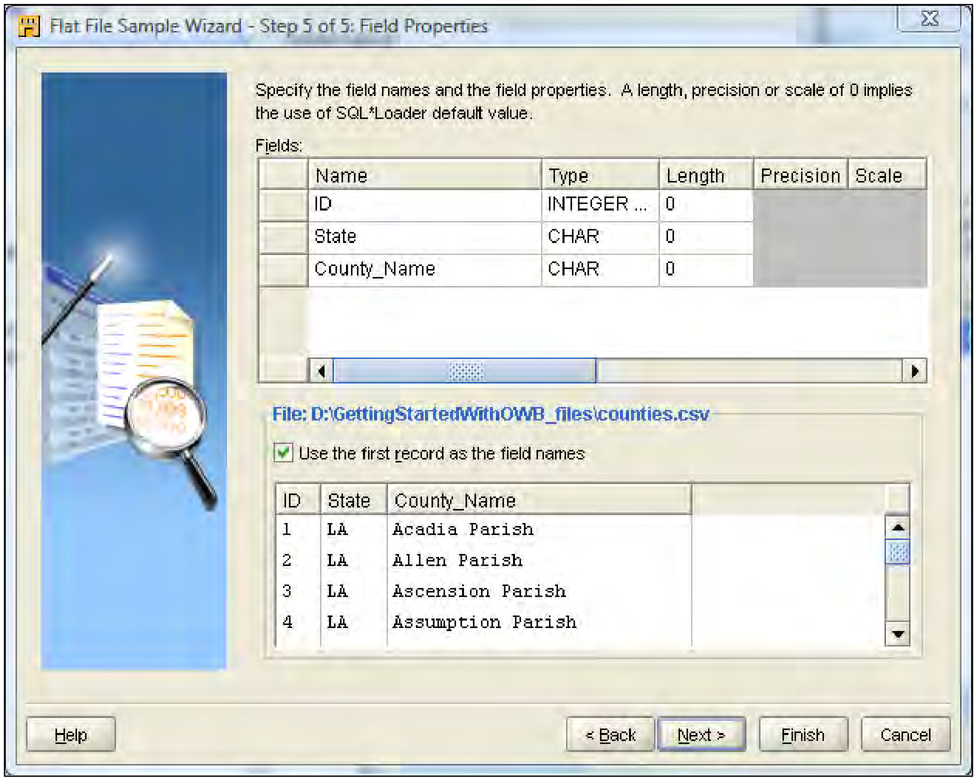
File: D:\GettingStartedWithOWB_files\counties.csv

☐ Use the first record as the field names

C1	C2	C3
ID	State	County Name
1	LA	Acadia Parish
2	LA	Allen Parish
3	LA	Ascension Parish

9. This is where we specify the details about what each field contains, and give each field a name. Here's where we can see why we didn't skip the first row in the previous step. Check the box that says **Use the first record as the field names** and we'll see that all the column names have changed to using the values from that first row. If we had skipped the first row, we would just have to manually enter a field name for each. That would not necessarily be a problem with our little sample file here, but it's not uncommon to receive a file with a large number of columns; and this can be a big time-saver. After clicking on the box, our screen now looks like the following screenshot:

Download at Boykma.Com



Specify the field names and the field properties. A length, precision or scale of 0 implies the use of SQL*Loader default value.

Fields:

Name	Type	Length	Precision	Scale
ID	INTEGER ...	0		
State	CHAR	0		
County_Name	CHAR	0		

File: D:\GettingStartedWithOWB_files\counties.csv

✓ Use the first record as the field names

ID	State	County_Name
1	LA	Acadia Parish
2	LA	Allen Parish
3	LA	Ascension Parish
4	LA	Assumption Parish

Notice that the field type for the first column has changed. The **ID** is now **INTEGER** instead of character, as it has now correctly detected that the remaining rows after that first one all contain integer data. **Length** is specified there, which defaults to **0**. If we scroll the window to the right, we'll also notice an SQL data type that is set for each field name. The reason for these extra attributes is that Warehouse Builder can directly use this file in a mapping or can use it indirectly referenced by an external table. An **external table** is a table definition within the Oracle database that actually refers to information stored in a flat file. The direct access is via the **SQL*Loader** utility. This is used to load files of information into a table in the database and when we specify the source in a mapping to be that file, it builds an SQL*Loader configuration to load it using the information provided here. More details about this can be found by clicking on the **Help** button on this screen. We do not need to worry about specifying a length here as the columns are delimited by commas. We can enter a value if we happen to know the maximum length a field could be.

Download at Boykma.Com

10. Click on **Next** to get a summary screen of what the wizard will do, or just click on the **Finish** button to continue. This will take us back to the **Import Metadata Wizard** screen where we can see that the red **X** has now changed to a green check mark. This indicates that we have specified all the information about the file.

What this **File Sample Wizard** just asked us for in these five steps is a direct example of what we mean when we talk about **metadata**. We just entered the data that describes our data contained in an imported file.

11. Click on the **Finish** button on the **Step 3** screen of the **Import Metadata Wizard**. It will create our file module under the **Files** node and we will be able to access it in the **Project Explorer**. We can see that the imported file is displayed as **COUNTIES_CSV**, which was the name it had defaulted to and which we left it set to.
12. We'll make sure to select **Save All** from the **Design** menu in Design Center to save the metadata we just entered.

This concludes all the object types we want to cover here for importing. Let's summarize what we've learned so far, and move on to the next step, which is defining our target.

Summary

That was a lot of information presented in this chapter. We began with a brief discussion about the source data for our scenario using the ACME Toys and Gizmos company. We then went through the process of importing metadata about our sources, and saw how to import metadata from an Oracle database as well as a flat file. Because we are dealing with a non-Oracle database, we were also exposed to configuring the Oracle database to communicate with a non-Oracle database using Oracle Heterogeneous Services. We also saw how to configure it for generic connectivity using the supplied ODBC agent. We worked through the process of manually creating table definitions for source tables for a SQL Server database.

At this point, you should run through these procedures to import or define the remaining tables that were identified in the source databases. For this, you can use the procedures we walked through above for practice. We'll be using the SQL Server database tables for the POS transactional database throughout the remainder of book, so be sure to have those defined at a minimum. Now that we have our sources all defined and imported, we need a target where we're going to load the data into from these sources. That's what we're going to discuss in the next chapter where we talk about designing and building our target data structures.

Download at Boykma.Com

3 Designing the Target Structure

We have our entire source structures defined in the Warehouse Builder. But before we can do anything with them, we need to design what our target data warehouse structure is going to look like. When we have that figured out, we can start mapping data from the source to the target. So, let's design our target structure. First, we're going to take a look at some design topics related to a data warehouse that are different from what we would use if we were designing a regular relational database. We'll then discuss what our design will look like, and after that we'll be ready to move right into creating that design using the Warehouse Builder in the next chapter.

Data warehouse design

When it comes to the design of a data warehouse, there is basically one option that makes the most sense for how we will structure our database and that is the **dimensional** model. This is a way of looking at the data from a business perspective that makes the data simple, understandable, and easy to query for the business end user. It doesn't require a database administrator to be able to retrieve data from it.

When looking at the source databases in the last chapter, we saw a normalized method of modelling a database. A normalized model removes redundancies in data by storing information in discrete tables, and then referencing those tables when needed. This has an advantage for a transactional system because information needs to be entered at only one place in the database, without duplicating any information already entered. For example, in the ACME Toys and Gizmos transactional database, each time a transaction is recorded for the sale of an item at a register, a record needs to be added only to the transactions table. In the table, all details regarding the information to identify the register, the item information, and the employee who processed the transaction do not need to be entered because that information is already stored in separate tables. The main transaction record just needs to be entered with references to all that other information.

Download at Boykma.Com

This works extremely well for a transactional type of system concerned with daily operational processing where the focus is on getting data into the system. However, it does not work well for a data warehouse whose focus is on getting data out of the system. Users do not want to navigate through the spider web of tables that compose a normalized database model to extract the information they need. Therefore, dimensional models were introduced to provide the end user with a flattened structure of easily queried tables that he or she can understand from a business perspective.

Dimensional design

A dimensional model takes the business rules of our organization and represents them in the database in a more understandable way. A business manager looking at sales data is naturally going to think more along the lines of "how many gizmos did I sell last month in all stores in the south and how does that compare to how many I sold in the same month last year?" Managers just want to know what the result is, and don't want to worry about how many tables need to be joined in a complex query to get that result. In the last chapter, we saw how many tables would have to be joined together in such a query just to be able to answer a question like the one above. A dimensional model removes the complexity and represents the data in a way that end users can relate to it more easily from a business perspective.

Users can intuitively think of the data for the above question as a cube, and the edges (or dimensions) of the cube labeled as stores, products, and time frame. So let's take a look at this concept of a cube with dimensions, and how we can use that to represent our data.

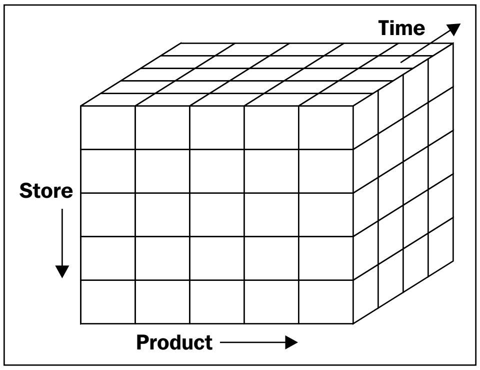
Cube and dimensions

The **dimensions** become the business characteristics about the sales, for example:

- A time dimension—users can look back in time and check various time periods
- A store dimension—information can be retrieved by store and location
- A product dimension—various products for sale can be broken out

Think of the dimensions as the edges of a cube, and the intersection of the dimensions as the measure we are interested in for that particular combination of time, store, and product. A picture is worth a thousand words, so let's look at what we're talking about in the following image:

Download at Boykma.Com



Notice what this cube looks like. How about a Rubik's Cube? We're doing a data warehouse for a toy store company, so we ought to know what a Rubik's cube is! If you have one, maybe you should go get it now because that will exactly model what we're talking about. Think of the width of the cube, or a row going across, as the product dimension. Every piece of information or measure in the same row refers to the same product, so there are as many rows in the cube as there are products. Think of the height of the cube, or a column going up and down, as the store dimension. Every piece of information in a column represents one single store, so there are as many columns as there are stores. Finally, think of the depth of the cube as the time dimension, so any piece of information in the rows and columns at the same depth represent the same point in time. The intersection of each of these three dimensions locates a single individual cube in the big cube, and that represents the measure amount we're interested in. In this case, it's dollar sales for a single product in a single store at a single point in time.

But one might wonder if we are restricted to just three dimensions with this model. After all, a cube has only three dimensions—length, width, and depth. Well, the answer is no. We can have many more dimensions than just three. In our ACME example, we might want to know the sales each employee has accomplished for the day. This would mean we would need a fourth dimension for employees. But what about our visualization above using a cube? How is this fourth dimension going to be modelled? And no, the answer is not that we're entering the Twilight Zone here with that "dimension not only of sight and sound but of mind..." We can think of additional dimensions as being cubes within a cube. If we think of an individual intersection of the three dimensions of the cube as being another cube, we can see that we've just opened up another three dimensions to use—the three for that inner cube. The Rubik's Cube example used above is good because it is literally a cube of cubes and illustrates exactly what we're talking about.

Download at Boykma.Com

We do not need to model additional cubes. The concept of cubes within cubes was just to provide a way to visualize further dimensions. We just model our main cube, add as many dimensions as we need to describe the measures, and leave it for the implementation to handle.

This is a very intuitive way for users to look at the design of the data warehouse. When it's implemented in a database, it becomes easy for users to query the information from it.

Implementation of a dimensional model in a database

We have seen how a dimensional model is preferred over a normalized model for designing a data warehouse. Now before we finalize our model for the ACME Toys and Gizmos data warehouse, let's look at the implementation of the model to see how it gets physically represented in the database. There are two options: a **relational** implementation and a **multidimensional** implementation. The relational implementation, which is the most common for a data warehouse structure, is implemented in the database with tables and foreign keys. The multidimensional implementation requires a special feature in a database that allows defining cubes directly as objects in the database. Let's discuss a few more details of these two implementations. But we will look at the relational implementation in greater detail as that is the one we're going to use throughout the remainder of the book for our data warehouse project.

Relational implementation (star schema)

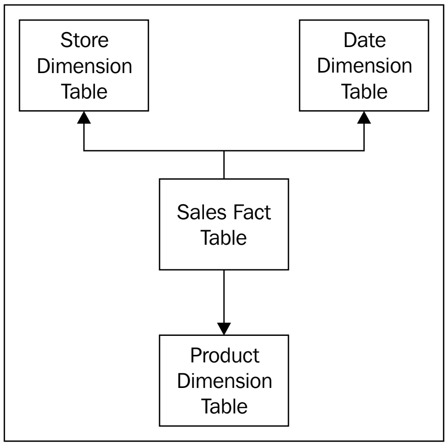
Back in Chapter 2, we saw how ACME's POS Transactional database and Order Entry databases were structured when we did our initial analysis. The diagrams presented showed all the tables interconnected, and we discussed the use of foreign keys in a table to refer to a row in another table. That is fundamentally a relational database. The term relational is used because the tables in it relate to each other in some way. We can't have a POS transaction without the corresponding register it was processed on, so those two relate to each other when represented in the database as tables.

Download at Boykma.Com

For a relational data warehouse design, the relational characteristics are retained between tables. But a design principle is followed to keep the number of levels of foreign key relationships to a minimum. It's much faster and easier to understand if we don't have to include multiple levels of referenced tables. For this reason, a data warehouse dimensional design that is represented relationally in the database will have one main table to hold the primary facts, or **measures** we want to store, such as *count of items sold* or *dollar amount of sales*. It will also hold descriptive information about those measures that places them in context, contained in tables that are accessed by the main table using foreign keys. The important principle here is that these tables that are referenced by the main table contain all the information they need and do not need to go down any more levels to further reference any other tables.

The ER diagram of such an implementation would be shaped somewhat like a star, and thus the term **star schema** is used to refer to this kind of an implementation. The main table in the middle is referred to as the **fact** table because it holds the facts, or measures that we are interested in about our organization. This represents the cube that we discussed earlier. The tables surrounding the fact table are known as dimension tables. These are the dimensions of our cube. These tables contain descriptive information, which places the facts in a context that makes them understandable. We can't have a dollar amount of sales that means much to us unless we know what item it was for, or what store made the sale, or any of a number of other pieces of descriptive information that we might want to know about it.

It is the job of data warehouse design to determine what pieces of information need to be included. We'll then design dimension tables to hold the information. Using the dimensions we referred to above in our cube discussion as our dimension tables, we have the following diagram that illustrates a star schema:



Download at Boykma.Com

Of course our star only has three points, but with a much larger data warehouse of many more dimensions, it would be even more star-like. Keep in mind the principle that we want to follow here of not using any more than one level of foreign key referencing. As a result, we are going to end up with a **de-normalized** database structure. We discussed normalization back in Chapter 2, which involved the use of foreign key references to information in other tables to lessen the duplication and improve data accuracy. For a data warehouse, however, the query time and simplicity is of paramount importance over the duplication of data. As for the data accuracy, it's a read-only database so we can take care of that up front when we load the data. For these reasons, we will want to include all the information we need right in the dimension tables, rather than create further levels of foreign key references. This is the opposite of normalization, and thus the term de-normalized is used.

Let's look at an example of this for ACME Toys and Gizmos to get a better idea of what we're talking about with this concept of de-normalization. Every product in our stores is associated with a department. If we have a dimension for product information, one of the pieces of information about the product would be the department it is in. In a normalized database, we would consider creating a department table to store department descriptions with one row for each department, and would use a short key code to refer to the department record in the product table.

However, in our data warehouse, we would include that department information, description and all, right in the product dimension. This will result in the same information being duplicated for each product in the department. What that buys us is a simpler structure that is easier to query and more efficient for retrieving information from, which is key to data warehouse usability. The extra space we consume in repeating the information is more than paid for in the improvement in speed and ease of querying the information. That will result in a greater acceptance of the data warehouse by the user community who now find it more intuitive and easier to retrieve their data.

In general, we will want to de-normalize our data warehouse implementation in all cases, but there is the possibility that we might want to include another level—basically a dimension table referenced by another dimension table. In most cases, we will not need nor want to do this and instances should be kept to an absolute minimum; but there are some cases where it might make sense.

This is a variation of the star schema referred to as a **snowflake schema** because with this type of implementation, dimension tables are partially normalized to pull common data out into secondary dimension tables. The resulting schema diagram looks somewhat like a snowflake. The secondary dimension tables are the tips of the snowflake hanging off the main dimension tables in a star schema.

Download at Boykma.Com

In reality, we'd want at the most only one or two of the secondary dimension tables; but it serves to illustrate the point. A snowflake dimension table is really not recommended in most cases because of ease-of-use and performance considerations, but can be used in very limited circumstances. The Kimball book on Dimensional Modelling was referred to at the beginning of Chapter 2. This book discusses some limited circumstances where it might be acceptable to implement a snowflake design, but it is highly discouraged for most cases.

Let's now talk a little bit about the multidimensional implementation of a dimensional model in the database, and then we'll design our cube and dimensions specifically for the ACME Toys and Gizmos Company data warehouse.

Multidimensional implementation (OLAP)

A multidimensional implementation or **OLAP** (**online analytic or analytical processing**) requires a database with special features that allow it to store cubes as actual objects in the database, and not just tables that are used to represent a cube and dimensions. It also provides advanced calculation and analytic content built into the database to facilitate advanced analytic querying. Oracle's Essbase product is one such database and was originally developed by Hyperion. Oracle recently acquired Hyperion, and is now promoting Essbase as a tool for custom analytics and enterprise performance management applications. The Oracle Database Enterprise Edition has an additional feature that can be licensed called **OLAP** that embeds a full-featured OLAP server directly in an Oracle database. This is an option organizations can leverage to make use of their existing database.

These kinds of analytic databases are well suited to providing the end user with increased capability to perform highly optimized analytical queries of information. Therefore, they are quite frequently utilized to build a highly specialized **data mart**, or a subset of the data warehouse, for a particular user community. The data mart then draws its data to load from the main data warehouse, which would be a relational dimensional star schema. A data warehouse implementation may contain any number of these smaller subset data marts.

We'll be designing dimensionally and implementing relationally, so let's now design our actual dimensions that we'll need for our ACME Toys and Gizmos data warehouse, and talk about some issues with the fact data (or cube) that we'll need. This will make the concepts we just discussed more concrete, and will form the basis for the work we do in the rest of the book as we implement this design. We'll then close out this chapter with a discussion on designing in the Warehouse Builder, where we'll see how it can support either of these implementations.

Download at Boykma.Com

We have seen the word dimension used in describing both a relational implementation and a multidimensional implementation. It is even in the name of the second implementation method we discussed, so why does the relational method use it also? In the relational case, the word is used more as an adjective to describe the type of table taken from the name of the model being implemented; whereas in the multidimensional model it's more a noun, referring to the dimension itself that actually gets created in the database. In both cases, the type of information conveyed is the same—descriptive information about the facts or measures—so its use in both cases is really not contradictory. There is a strong correlation between the fact table of the relational model and the cube of the dimensional model, and between the dimension tables of the relational model and the dimensions of the dimensional model.

Designing the ACME data warehouse

We have chosen to use a dimensional model for our data warehouse, so we'll define a cube with dimensions to represent our information. Let's lay out a basic structure of information we want each to contain. We'll begin with the dimensions, since they are going to provide the context for the measure(s) we will want to store in our cube.

Identifying the dimensions

To know what dimensions to design for, we need to know what business process we're going to be supporting with our data warehouse. Is management concerned with daily inventory? How about daily sales volume? This information will guide us in selecting the correct parts of the business to model with our dimensions.

We are going to support the sales managers in managing the daily sales of the ACME Toys and Gizmos Company, and they have already given us an example of the kind of question they want answered from their data warehouse, as we saw earlier. We used that to illustrate the cube concept and to show a star schema representation of it, so the information shows us the dimensions we need. Since management is concerned with daily sales, we need some kind of date/time dimension that will provide us the context for the sales data indicating what day the sale transaction took place.

We can pretty much be guaranteed that we will need a time/date type dimension for any data warehouse we design, since one of the main features of data warehouses is to provide time-series type analytical query capabilities (as we talked about earlier).

Download at Boykma.Com

Are we going to need both the time and the date in this dimension, or will just the date be sufficient? We can get an answer to this question by also looking back at our business process, which showed that management is concerned with daily sales volume. Also, the implementation of the time dimension in OWB does not include the time of day since it would have to include 24 hours of time values for each day represented in the dimension due to the way it implements the dimension. In the future if time is needed, there are options for creating a separate dimension just for modelling time of day values. For our initial design we'll call our time related dimension a Date dimension just for added clarity.

Another dimension we have included is to model the product information. Each sale transaction is for a particular product, and management has indicated they are concerned about seeing how well each product is selling. So we will include a dimension that we shall call Product. At a minimum we need the product name, a description of the product, and the cost of the product as attributes of our product dimension—so we'll include those in our logical model.

So far we have a Date dimension to represent our time series and a Product dimension to represent the items that are sold. We could stop there. Management would then be able to query for sales data for each day for each product sold by ACME Toys and Gizmos, but they wouldn't be able to tell where the sale took place. Another key piece of information the management would like to be able to retrieve is how well the stores are doing compared to each other for daily sales. Unless we include some kind of a location dimension, they will not be able to tell that. That is why we have included a third dimension called Store. It is used to maintain the information about the store that processed the sales transaction. For attributes of the store dimension, we can include the store name and address at a minimum to identify each store.

These dimensions should be enough to satisfy the management's need for querying information for this particular business process—the daily sales. We could certainly include a large number of other dimensions, but we'll stop here to keep this simple for our first data warehouse. We can now consider designing the cube and what information to include in it.

Designing the cube

In the case of the ACME Toys and Gizmos Company, we have seen that the main measure the management is concerned about is daily sales. There are other numbers we could consider such as inventory numbers: How much of each item is on hand? However, the inventory is not directly related to daily sales and wouldn't make sense here. We can model an inventory system in a data warehouse that would be separate from the sales portion. But for our purpose, we're going to model the sales. Therefore, our main measure is going to be the dollar amount of sales for each item.

Download at Boykma.Com

A very important topic to consider at this point is what will be the **grain** of the measure—the sales data—that we're going to store in our cube? The grain (or **granularity**) is the level that the sales number refers to. Since we're using sales as the measure, we'll store a sales number; and from our dimensions, we can see that it will be for a given date in a given store and for a given product. Will that number be the total of all the sales for that product for that day? Yes, so it satisfies our design criteria of providing daily sales volume for each product. That is the smallest and lowest level of sales data we want to store. This is what we mean by the grain or granularity of the data.

Levels/hierarchies

A dimensional model is naturally able to handle this concept of the different levels of data by being able to model a hierarchy within a dimension. The time/date dimension is an easy example of using of various levels. Add up the daily totals to get the totals for the month, and add up 12 monthly totals to get the yearly sales. The time/date dimension just needs to store a value to indicate the day, month, and year to be able to provide a view of the data at each of those levels. Combining various levels together then defines a hierarchy. By storing data at the lowest level, we make available the data for summing at higher levels. Likewise, from a higher level, the data is then available to drill down to view at a lower level. If we were to arbitrarily decide to store the data at a higher level, we would lose that flexibility. We'll discuss this further in the next chapter when we build our time dimension in the Warehouse Builder.

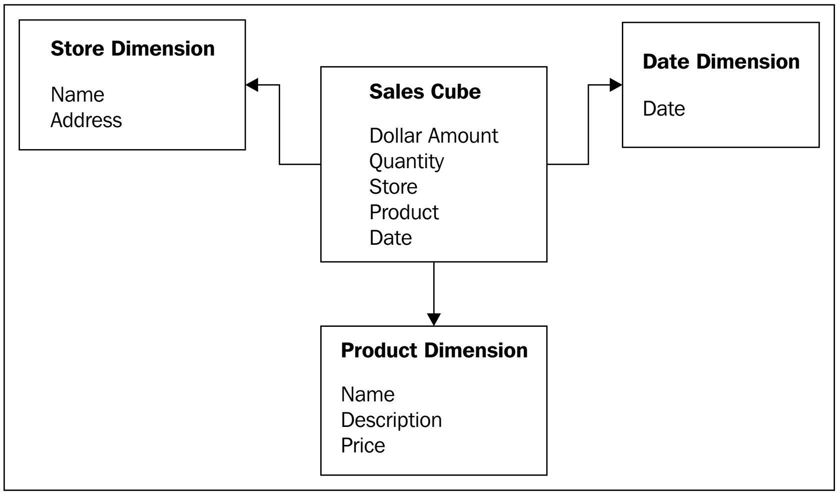
In this case, we have a source system—the POS Transactional system—that maintains the dollar amount of sales for each line item in each sales transaction that takes place. This can provide us the level of detail we will want to capture and maintain in our cube, since we can definitely capture sales for each product at each store for each day. We have found out that the POS Transactional system also maintains the count of the number of a particular item sold in the transaction. This is an additional measure we will consider storing in our cube also, since we can see that it is at the same grain as the total sales. The count of items would still pertain to that single transaction just like the sales amount, and can be captured for each product, store, and even date.

The only other pieces of information our cube is going to contain are pointers to the dimensions. In the relational model, the fact table would contain columns for the dollar amount, the quantity, the unit cost, and then foreign keys for each of the dimension tables.

Download at Boykma.Com

There are times when it's valid in dimensional design to include more descriptive information right in the cube, rather than create a dimension for it. There may be some particularly descriptive piece of information that stands all by itself, which is not associated with anything else or whose additional descriptive information has already been included in other dimensions. In that case, it wouldn't make sense to create a whole dimension just for it; so it is included directly in the fact table or cube. This is referred to as a degenerate dimension. It is explained in more detail in the Kimball book on dimensional modelling we talked about earlier. There are many other aspects to dimensional design that we don't have the space to cover here, but are covered in the Kimball book in more detail. It would be a good idea for you to read this book or a similar one to get a better understanding of the detailed dimensional modelling concepts such as this.

Our design is drawn out in a star schema configuration showing the cube, which is surrounded by the dimensions with the individual items of information (attributes) we'll want to store for each. It looks like the following:

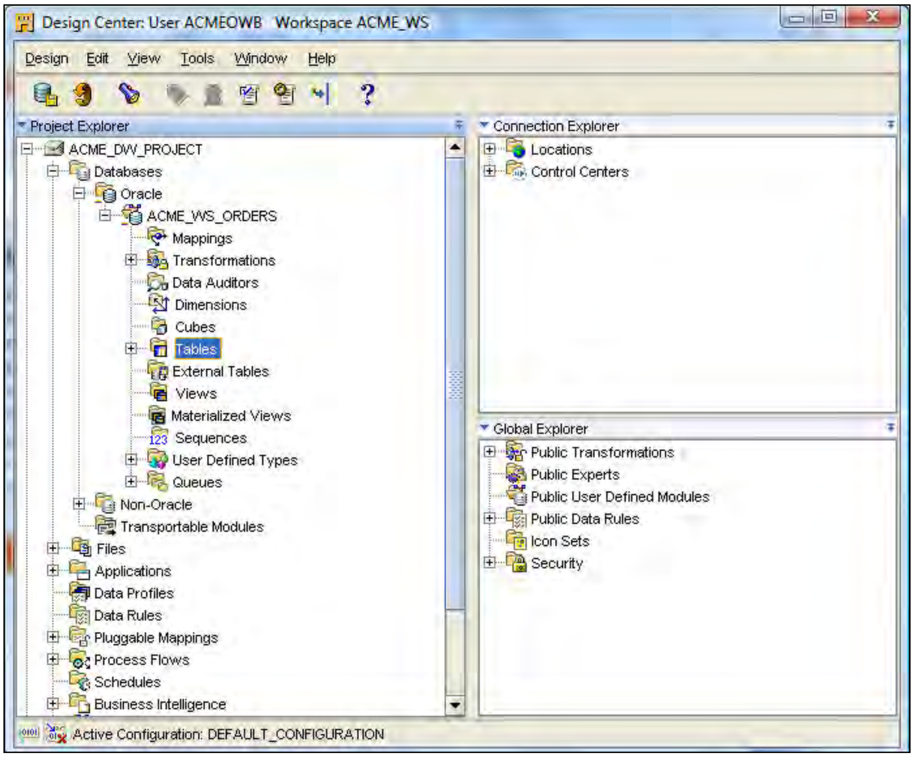


OK, we now have a design for our data warehouse. It's time to see how OWB can support us in entering that design and generating its physical implementation in the database.

Download at Boykma.Com

Data warehouse design in OWB

The Warehouse Builder contains a number of objects, which we can use in designing our data warehouse, that are either relational or dimensional. OWB currently supports designing a target schema only in an Oracle database, and so we will find the objects all under the **Oracle** node in the **Project Explorer**. Let's launch **Design Center** now and have a look at it. But before we can see any objects, we have to have an **Oracle** module defined to contain the objects. If you've been following along and working through the examples in this book, so far you should have one module already defined for the ACME web site orders database—`ACME_WS_ORDERS`. We created this in the last chapter when we imported our metadata from that source. If that is the case, our **Project Explorer** window will look similar to the following:



Creating a target user and module

We need a different module to create our target objects in. So before going any further, let's create a new module in the **Project Explorer** for our target to hold our data warehouse design objects. However, before we can do that, we should have a target schema defined in the database that will hold our target objects when we deploy them.

Download at Boykma.Com

So far we have discussed many different components such as the repository, workspaces, the design center, and so on. So, it can be confusing to know exactly where our main data warehouse is going to be located. The *target schema* is going to be the main location for the data warehouse. When we talk about our "data warehouse" after we have it all constructed and implemented, the target schema is what we will be referring to. Amid all these different components we discussed that compose the Warehouse Builder, the target is where the actual data warehouse will be built. Our design will be implemented there, and the code will be deployed to that schema by OWB to load the target structure with data from the sources.

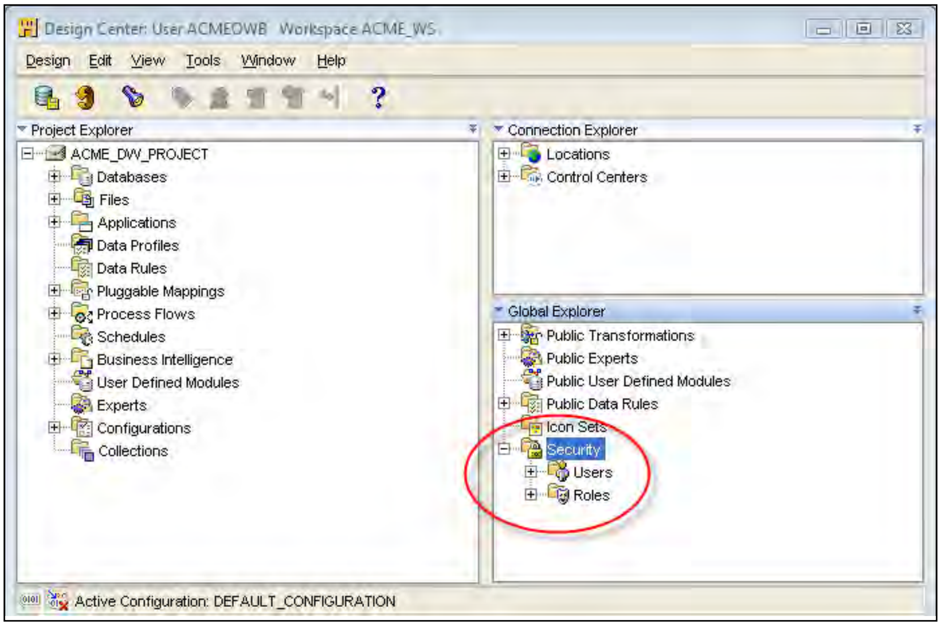
Every target module must be mapped to a target user schema. Back in Chapter 1, when we ran the Repository Assistant to create the repository and workspace, we created the **acmeowb** user as the repository owner and mentioned that this user can be a deployment target for our data warehouse. However, it does not have to be the target user. It's a good idea to create a separate user schema to become the target so that user roles in our database can be kept separate. Using the OWB repository owner schema would mean our target data warehouse would have to be on the same database server as our repository. In large installations, that will most likely not be the case. So for maximum flexibility, we're going to create a separate user schema. In our case, that user will be created in the same database as the repository; but it can be moved to another database easily if we expand and add more servers.

Create a target user

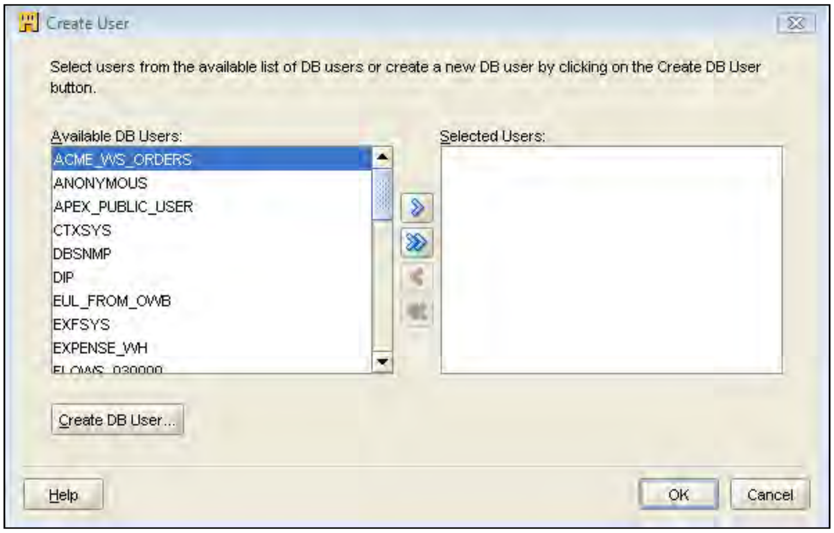
There are a couple of ways we can go about creating our target user—create the user directly in the database and then add to OWB, or use OWB to physically create the user. If we have to create a new user, and if it's on the same database as our repository and workspaces, it's a good idea to use OWB to create the user, especially if we are not that familiar with the SQL command to create a user. However, if our target schema were to be in another database on another server, we would have to create the user there. It's a simple matter of adding that user to OWB as a target, which we'll see in a moment. Let's begin in the **Design Center** under the **Global Explorer**. We talked about that **Global Explorer** back in our introduction to the **Design Center** in Chapter 2. There we said it was for various objects that pertained to the workspace as a whole.

Download at Boykma.Com

One of those object types is a **Users** object that exists under the **Security** node as shown here:



Right-click on the **Users** node and select **New...** to launch the **Create User** dialog box as shown here:


Create User

Select users from the available list of DB users or create a new DB user by clicking on the Create DB User button.

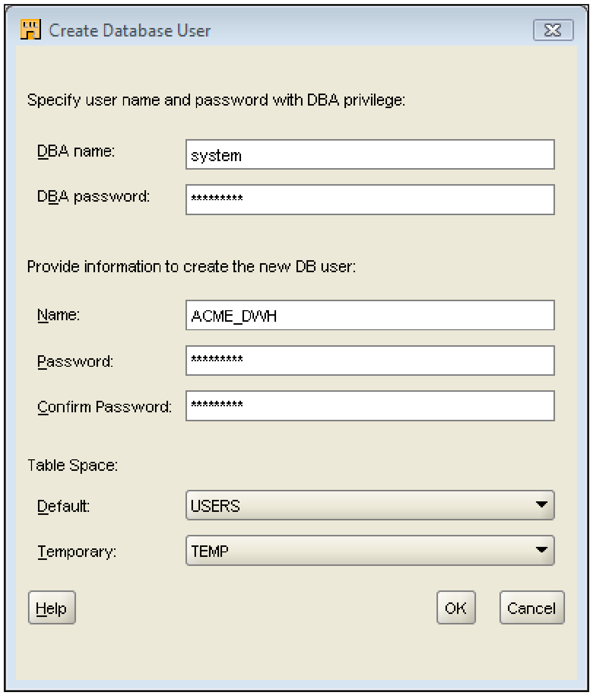
Available DB Users:
ACME_WS_ORDERS
ANONYMOUS
APEX_PUBLIC_USER
CTXSYS
DBSNMP
DIP
EUL_FROM_OWB
EXFSYS
EXPENSE_WH
FLOWS_030000

Selected Users:

Download at Boykma.Com

With this dialog box, we are creating a workspace user. We create a workspace user by selecting a database user that already exists or create a new one in the database. If we already had a target user created in the database, this is where we would select it. We're going to click on the **Create DB User...** button to create a new database user.

We need to enter the **system** username and password as we need a user with DBA privileges in the database to be able to create a database user. We then enter a username and password for our new user. As we like to keep things basic, we'll call our new user **ACME_DWH**, for the ACME data warehouse. We can also specify the default and temporary tablespace for our new user, which we'll leave at the defaults. The dialog will appear like the following when completely filled in:


Create Database User

Specify user name and password with DBA privilege:

DBA name:	system
DBA password:	*********

Provide information to create the new DB user:

Name:	ACME_DWH
Password:	*********
Confirm Password:	*********

Table Space:

Default:	USERS
Temporary:	TEMP

The new user will be created when you click on the **OK** button, and will appear in the righthand window of the **Create User** dialog already selected for us. Click on the **OK** button and the user will be registered with the workspace, and we'll see the new username if we expand the **Users** node under **Security** in the **Global Explorer**. We can continue with creating our target module now that we have a user defined in the database to map to.

Download at Boykma.Com

Notice that the previous dialog boxes don't have any way to specify the database location information. This is because it creates the user on the local database we were connected to when we logged into the **Design Center**, which is the location of our repository and workspaces. Due to this, this method can only be used to create the user if it is on the local database. In the next section where we create our target module, we'll get to specify the location and that dialog box will allow us to specify a remote database if needed.

Create a target module

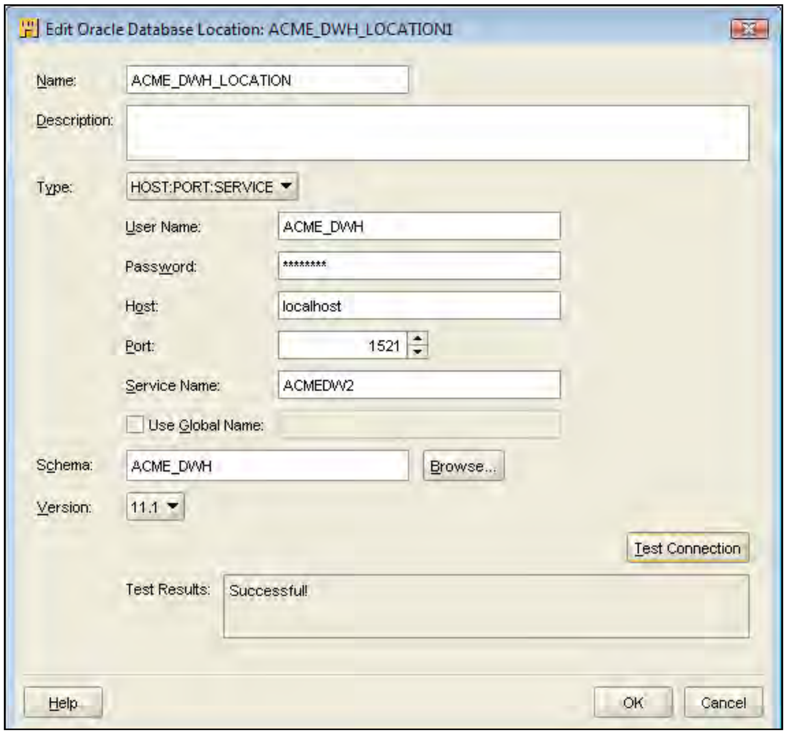
We'll follow the same steps as we did in the last chapter where we created the `ACME_WS_ORDERS` module. Right-click on the **Oracle** object under **Databases** and select **New...** from the pop-up menu to launch the **Create Module Wizard** and step through the process. We'll name this module `ACME_DWH` for ACME Data Warehouse.

An important difference between creating this module and the module we created in the last chapter is that we need to designate this one as a target module. On the **Step 1 of 2** screen, be sure to select **Warehouse Target** as the type rather than **Data Source**.

Create the location in the next step, which is similar to how we created it in the last chapter, but be sure to specify the database that is our target database instead of the source we used last time. If we're creating our own test system, the source location may very well be the same as our target. But in real-world situations, it will likely be in a different database on a different server. If we had created a target user schema on a different database, this is the point at which we would be able to enter the connection information for that user in order to associate our target module with that user and make it a target.

Download at Boykma.Com

For reference, the location screen should look like the following for defining the location of the target module:


Edit Oracle Database Location: ACME_DWH_LOCATION1

Name:	ACME_DWH_LOCATION	
Description:		
Type:	HOST:PORT:SERVICE	
	User Name:	ACME_DWH
	Password:	********
	Host:	localhost
	Port:	1521
	Service Name:	ACMEDW2
	Use Global Name:	
Schema:	ACME_DWH	
Version:	11.1	
	Test Results:	Successful!

We have specified the **User Name** to be the user we just created for this very purpose in the previous section.

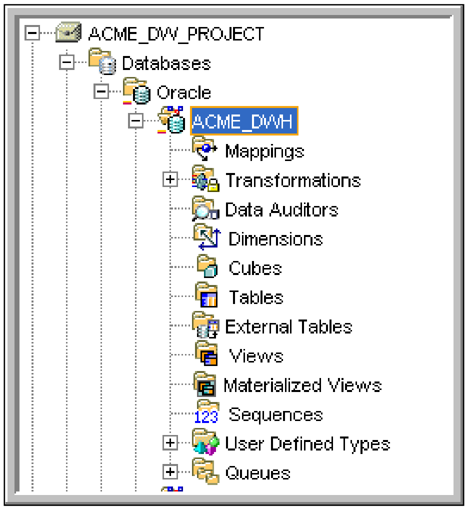
In this dialog box, we can see the location information (**Host**, **Port** and **Service Name**) that we can use to specify a user in another database if needed. If our user were not in this database, we would just enter his or her appropriate host and port for the location and the service name of that remote database.

Now that we have our target database schema and a target module defined, which is associated with a location pointing to that target schema, we will now have two Oracle modules under our Oracle object in Project Explorer. We can continue our discussion of the design objects available to us in the Warehouse Builder for designing our database. First, let's make sure we save our work so far by using the *Ctrl+S* key combination or by selecting **Design | Save All** from the main menu.

Download at Boykma.Com

OWB design objects

Looking at our **Project Explorer** window with our target Oracle module expanded, we can see a number of objects that are available to us as shown here:



There are objects that are relational such as **Tables**, **Views**, **Materialized Views**, and **Sequences**. Also, there are dimensional objects such as **Cubes** and **Dimensions**. We just discussed relational objects versus dimensional objects. We have decided to model our database dimensionally and this will dictate the objects we create. From the standpoint of providing the best model of our business rules and representing what users want to see, the dimensional method is the way to go as we already discussed. Most data warehouse implementations we encounter will use a dimensional design. It just makes more sense for matching the business rules the users are familiar with and providing the types of information the user community will want to extract from the database.

We are thinking dimensionally in our design, but what about the underlying physical implementation? We discussed the difference between the relational and multidimensional physical implementation of a database, and now it's time to see how we will handle that here. The Warehouse Builder can help us tremendously with that because it has the ability to design the objects logically using cubes and dimensions in a dimensional design. It also has the ability to implement them physically in the underlying database as either a relational structure or a dimensional structure simply by checking a box.

Download at Boykma.Com

In general, which option should be chosen? The relational implementation is best suited to large amounts of data that tend to change more frequently. For this reason, the relational implementation is usually chosen for the main data warehouse schema by most implementers of a data warehouse. It is much better suited to handling the large volumes of data that are imported frequently into the data warehouse. The multidimensional implementation is better suited to applications where heavy analytic processing is required, and so is a good candidate for the data marts that will be presented to users.

To be able to implement the design physically as a dimensional implementation with cubes and dimensions, we need a database that is designed specifically to support **OLAP** as we discussed previously. If that is not available, then the decision is made for us. In our case, when we installed the Oracle database in Chapter 1, we installed the Enterprise Edition with default options, and that includes the OLAP feature in the database, so we have a choice to make. Since we're installing our main data warehouse target schema, we'll choose the relational implementation.

For a relational implementation, the Warehouse Builder actually provides us two options for implementing the database: a pure relational option and the relational OLAP option. If we were to have the OLAP feature installed in our database, we could choose to still have the cubes and dimensions implemented physically in a relational format. We could have it store metadata in the database in the OLAP catalog, and so multidimensional features such as aggregations would be available to us. We could take advantage of the relational implementation of the database for handling large volumes of data, and still implement a query or reporting tool such as Oracle Discoverer to access the data that made use of the OLAP features. The pure relational option just depends on whether we choose to deploy only the data objects and not the OLAP metadata. In reality, most people choose either the pure relational or the multidimensional. If they want both, they implement separate data marts. In fact, the default when creating dimensional objects and selecting relational for the implementation is to only deploy data objects.

Just to be clear, does all this mean that if we haven't paid for the OLAP feature for our database, we can only design our data warehouse using the relational objects; and therefore must our decision to design dimensionally change? The answer to that would be an emphatic *no*, since we just mentioned how OWB will let us design dimensional objects, cubes and dimensions, and then implement them physically in the database as relational objects. The benefit is that the same dimensional design can be implemented at a later time in an OLAP database just by changing a single setting. There are features of the Warehouse Builder for handling dimensional features automatically for us, such as levels, surrogate keys, and slowly changing dimensions (all of which we'll talk about later) that designing dimensionally

Download at Boykma.Com

provides us. We would have to implement these manually if we designed our own tables. Most people who use the Warehouse Builder will use it in that way, so we'll definitely want to make use of that feature to maximize the usefulness of the tools to us. This provides us with flexibility and it is the way we are going to proceed with our design. We'll design dimensionally using a cube and dimensions, and then can implement it either relationally or dimensionally when we're ready.

Summary

We have now gone through the process of designing the target structure for our data warehouse. We began with a very high-level overview of data warehouse design topics, then talked about dimensional design and the relational versus multidimensional implementation, and then we discussed the differences between them. As was mentioned earlier, there are other books that are devoted solely to this topic and it would be good to read one or more of them to learn more about design than we've been able to cover here. Our design for ACME Toys and Gizmos is very rudimentary, just to give us an introduction to designing in OWB. You'll want to read in more detail about design when you tackle a real-world design because you may run into other issues we didn't have time or space to cover here.

We're going to take this up now and actually implement the design in OWB in the next chapter.

Download at Boykma.Com

4
Creating the Target Structure in OWB

Now it's time to actually start creating objects in the Warehouse Builder for our target structure. In the previous chapter, we decided what our cube and dimensions were going to be in our logical design and now we are at the point where we can implement that design in OWB. We'll create the objects using the wizards that the Warehouse Builder provides for us to simplify the task of building cubes and dimensions. We'll look at the Data Object Editor in a little more detail than we saw in Chapter 2. Let's begin with creating the dimensions.

Creating dimensions in OWB

The Warehouse Builder provides a couple of ways to create a dimension. One way is to use the wizards that it provides, which will automatically create a dimension for us. The other way is to manually create it. We have identified three dimensions that we are going to need—a Date dimension, a Product dimension, and a Store dimension. The Date dimension, as we've seen, is our time/date dimension for providing a time series for our data. That kind of dimension is common to most data warehouses and the information it contains is very similar from warehouse to warehouse. So, recognizing this commonality, the Warehouse Builder provides us a special wizard to use just for time dimensions. Let's begin with that one.

Let's talk a bit about "creating". Throughout this chapter we'll discuss creating objects, but what we're really creating is the metadata that describes the objects. Nothing will be actually created in the database yet. We won't actually do that until Chapter 8 when we deploy our design to the target schema.

Download at Boykma.Com

The Time dimension

Let's discuss briefly what a Time dimension is, and then we'll dive right into the Warehouse Builder Design Center and create one. A Time dimension is a key part of most data warehouses. It provides the time series information to describe our data. A key feature of data warehouses is being able to analyze data from several time periods and compare results between them. The Time dimension is what provides us the means to retrieve data by time period.

> Do not be confused by the use of the word *Time* to refer to this dimension. In this case, it does not refer to the time of day but to time in general which can span days, weeks, months, and so on. We are using it because the Warehouse Builder uses the word *Time* for this type of dimension to signify a time period. So when referring to a *Time* dimension here, we will be talking about our time period dimension that we will be using to store the date. We will give the name `Date` to be clear about what information it contains.

Every dimension, whether time or not, has four characteristics that have to be defined in OWB:

- **Levels**
- **Dimension Attributes**
- **Level Attributes**
- **Hierarchies**

The Levels are for defining the levels where aggregations will occur, or to which data can be summed. We must have at least two levels in our Time dimension. While reporting on data from our data warehouse, users will want to see totals summed up by certain time periods such as per day, per month, or per year. These become the levels. A multidimensional implementation includes metadata to enable aggregations automatically at those levels, if we use the OLAP feature. The relational implementation can make use of those levels in queries to sum the data. The Warehouse Builder has the following Levels available for the Time dimension:

- Day
- Fiscal week
- Calendar week
- Fiscal month
- Calendar month

Download at Boykma.Com

- Fiscal quarter
- Calendar quarter
- Fiscal year
- Calendar year

The Dimension Attributes are individual pieces of information we're going to store in the dimension that can be found at more than one level. Each level will have an **ID** that identifies that level, a **start and an end date** for the time period represented at that level, a **time span** that indicates the number of days in the period, and a **description** of the level.

Each level has Level Attributes associated with it that provide descriptive information about the value in that level. The dimension attributes found at that level and additional attributes specific to the level are included. For example, if we're talking about the **Month** level, we will find attributes that describe the value for the month such as the month of the year it represents, or the month in the calendar quarter. These would be numbers indicating which month of the year or which month of the quarter it is.

The Oracle Warehouse Builder Users' Guide contains a more complete list of all the attributes that are available. OWB tracks which of these attributes are applicable to which level and allows the setting of a separate description that identifies the attribute for that level. Toward the end of the chapter, when we look at the **Data Object Editor**, we'll see the feature provided by the Warehouse Builder to view details about objects such as dimensions and cubes.

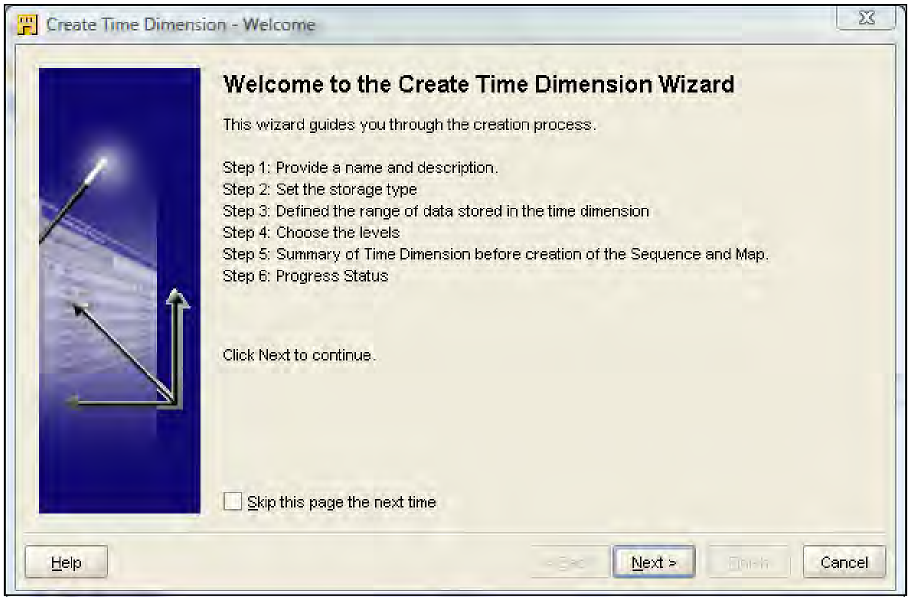
We must also define at least one **Hierarchy** for our Time dimension. A hierarchy is a structure in our dimension that is composed of certain levels in order; there can be one or more hierarchies in a dimension. Calendar month, calendar quarter, and calendar year can be a hierarchy. We could view our data at each of these levels, and the next level up would simply be a summation of all the lower-level data within that period. A calendar quarter sum would be the sum of all the values in the calendar month level in that quarter, and the multidimensional implementation includes the metadata to facilitate these kinds of calculations. This is one of the strengths of a multidimensional implementation.

The good news is that the Warehouse Builder contains a wizard that will do all the work for us—create our Time dimension and define the above four characteristics—just by asking us a few questions.

Download at Boykma.Com

Creating a Time dimension with the Time Dimension Wizard

Let's start creating our Time dimension by launching **Design Center** if it's not already running. In the **Project Explorer** window, we're going to expand the **Databases** node under `ACME_DW_PROJECT`, and then our ACME data warehouse node `ACME_DWH`. We will right-click on the **Dimensions** node, and select **New | Using Time Wizard...** to launch the **Time Dimension Wizard**. The **Time Dimension Wizard** will walk us through a six-step process to define the characteristics of our Time dimension. The first screen will describe these steps for us, which is shown here so we can see what it will be asking us:


Create Time Dimension - Welcome
Welcome to the Create Time Dimension Wizard

This wizard guides you through the creation process.

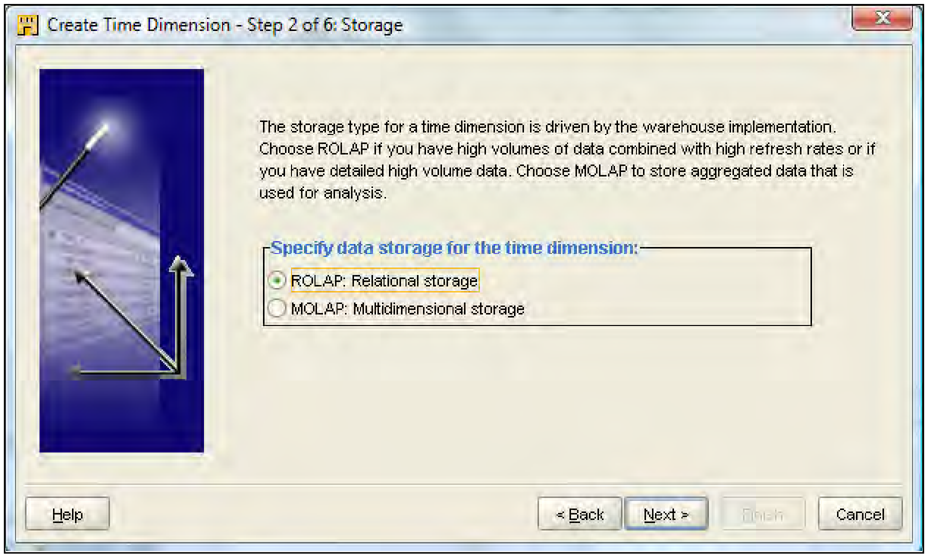
Step 1: Provide a name and description.
Step 2: Set the storage type
Step 3: Defined the range of data stored in the time dimension
Step 4: Choose the levels
Step 5: Summary of Time Dimension before creation of the Sequence and Map.
Step 6: Progress Status

Click Next to continue.

Skip this page the next time
Help
< Back
Next >
Finish
Cancel

1. The first step of the wizard will ask us for a name for our Time dimension. We're going to call it `DATE_DIM`. If we try to use just DATE, it will give us an error message because that is a reserved word in the Oracle Database; so it won't let us use it.
2. The next step will ask us what type of storage to use for our new dimension, as shown here:

Download at Boykma.Com


Create Time Dimension - Step 2 of 6: Storage

The storage type for a time dimension is driven by the warehouse implementation. Choose ROLAP if you have high volumes of data combined with high refresh rates or if you have detailed high volume data. Choose MOLAP to store aggregated data that is used for analysis.

Specify data storage for the time dimension:

[✓ ROLAP: Relational storage]

MOLAP: Multidimensional storage

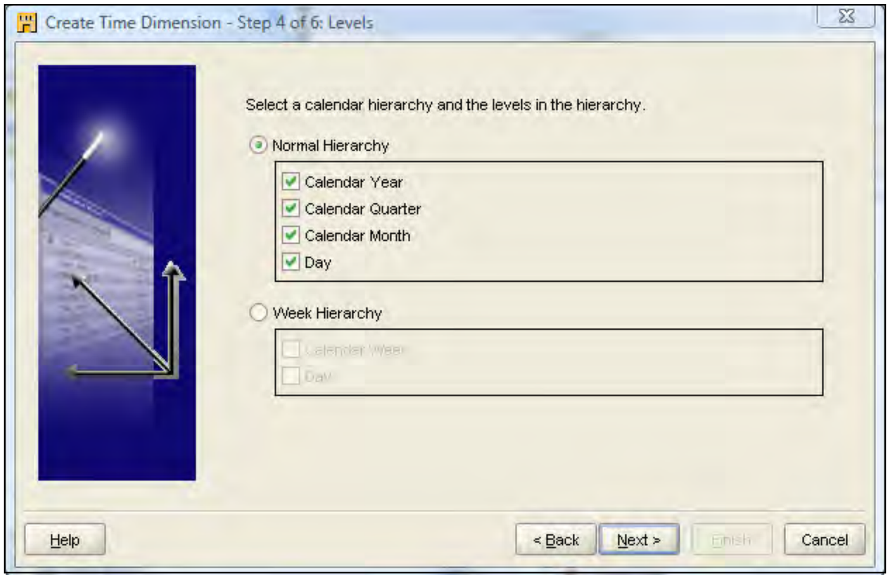
Here we get to designate whether we want a relational physical implementation in the database or a multidimensional implementation. This is what was referred to earlier as checking a box to switch between the two. Simply select one or the other, and this is how our design will be implemented in the database with no changes by us required at all. As we discussed in the last chapter, we're going to implement our data warehouse using the pure relational option. So we're going to select ROLAP, as shown in the image above. Both the pure relational implementation and the relational OLAP option, which we discussed in the last chapter, are available by selecting the ROLAP option here. We can set a deployment configuration option that defaults to deploying data objects only. But this can be changed to deploy the OLAP metadata to the OLAP catalog also. In both cases, this will result in the generation of relational database objects in a star schema. However, if that option is selected, it will only store the OLAP metadata in the OLAP catalog in the database. We'll see where to set that option when we look at the Data Object Editor.

Download at Boykma.Com

3. Now this brings us to step 3, which asks us to specify the data generation information for our dimension. The Time Dimension Wizard will be automatically creating a mapping for us to populate our Time dimension and will use this information to load data into it. It asks us what year we want to start with, and then how many total years to include starting with that year. The numbers entered here will be determined by what range of dates we expect to load the data for, which will depend on how much historical data we will have available to us. We have checked with the DBAs for ACME Toys and Gizmos Company to get an idea of how many years' worth of data they have and have found out that there is data going back three years. Based on this information, we're going to set the start year to 2007 with the number of years set to three to bring us up to 2009.

   The other option available to us on the data generation step is the type of Time dimension to create. It can be based on a calendar year or fiscal year. This provides us with the flexibility to define our Time dimension based on what our company actually uses for its financial year. ACME Toys and Gizmos Company operates on a calendar-year basis, so we'll leave it set at calendar.

4. This step is where we choose the hierarchy and levels for our Time dimension. We have to select one of the two hierarchies. We can use the **Normal Hierarchy** of day, month, quarter, and year; or we can choose the **Week Hierarchy**, which consists of two levels only—the day and the calendar week. Notice that if we choose the **Week Hierarchy**, we won't be able to view data by month, quarter, or year as these levels are not available to us. This is seen in the following image:



Create Time Dimension - Step 4 of 6: Levels

Select a calendar hierarchy and the levels in the hierarchy.

[✓ Normal Hierarchy]
- [✓] Calendar Year
- [✓] Calendar Quarter
- [✓] Calendar Month
- [✓] Day

Week Hierarchy
- [ ] Calendar Week
- [ ] Day

Download at Boykma.Com

The levels are not available to us because a week does not roll up or aggregate to a month. Some months have four weeks while some have five, and that's not even exact weeks. The only month that has a month evenly divided by weeks is February, and that's only during non-leap years. So, we can see that weeks do not sum up nicely into months, or any higher level of time. How about year? Surely, that must sum up nicely we might say, aren't there 52 weeks in a year? Multiply 52 by 7 and we get 364 days. So, even that won't work. Thus, if we choose to model weeks as one of our levels, we get day and week and that's it.

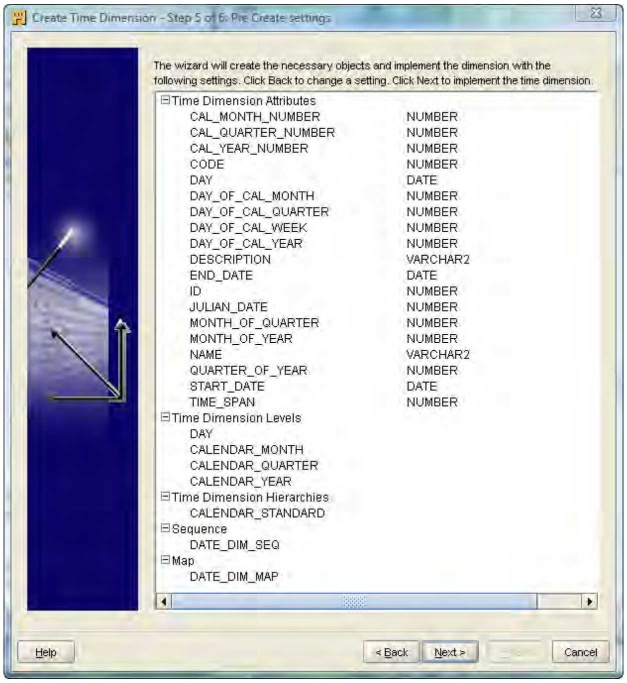
[This points out an important aspect of aggregation when deciding what our levels should be. It's very important to keep that idea of aggregation or summing in mind when choosing levels, or we will end up with data that doesn't make sense. The **Time Dimension Wizard** will not allow us to choose levels that don't sum up correctly because it has predefined a list of levels for us to choose from, with preset hierarchies. However, when defining any other dimension type, we'll definitely have to keep this in mind as we'll be specifying levels and hierarchies ourselves rather than choosing from the predefined ones.]

We're going to select the normal hierarchy, and now we can choose which of the levels to include. It is always a good idea to include the lowest level possible in our hierarchy to provide maximum flexibility in aggregating data in this dimension. If we leave out day, then we will never be able to view our data by day, but only by month at the lowest level.

5. Let's move on to step 5 where the wizard will provide us the details about what it is going to create. An example is shown in the following image, which is what you should see if you've made all the same selections as we've moved along.

Download at Boykma.Com

In the following image we can see the dimension attributes, levels, and hierarchies that will be created:



The wizard will create the necessary objects and implement the dimension with the following settings. Click Back to change a setting. Click Next to implement the time dimension.

```
⊟Time Dimension Attributes
    CAL_MONTH_NUMBER          NUMBER
    CAL_QUARTER_NUMBER        NUMBER
    CAL_YEAR_NUMBER           NUMBER
    CODE                      NUMBER
    DAY                       DATE
    DAY_OF_CAL_MONTH          NUMBER
    DAY_OF_CAL_QUARTER        NUMBER
    DAY_OF_CAL_WEEK           NUMBER
    DAY_OF_CAL_YEAR           NUMBER
    DESCRIPTION               VARCHAR2
    END_DATE                  DATE
    ID                        NUMBER
    JULIAN_DATE               NUMBER
    MONTH_OF_QUARTER          NUMBER
    MONTH_OF_YEAR             NUMBER
    NAME                      VARCHAR2
    QUARTER_OF_YEAR           NUMBER
    START_DATE                DATE
    TIME_SPAN                 NUMBER
⊟Time Dimension Levels
    DAY
    CALENDAR_MONTH
    CALENDAR_QUARTER
    CALENDAR_YEAR
⊟Time Dimension Hierarchies
    CALENDAR_STANDARD
⊟Sequence
    DATE_DIM_SEQ
⊟Map
    DATE_DIM_MAP
```

We can also see a couple of extra items at the bottom that we haven't discussed yet such as a sequence and a map name.

The sequence is an object that will be created to populate the ID values with unique numbers. It is created automatically for us by the wizard. This ID value is used as what is called the **Surrogate Identifier** for a level record. This value stands in (acts as a surrogate) for the actual unique identifier for the record. The actual identifier is called a **Business Identifier**. It contains one or more attributes that have been selected by us to uniquely represent a one-level record to differentiate it from another.

Download at Boykma.Com

When we link a dimension to a cube, it will use that surrogate identifier as the key to link to, as this is easier for the database to use than a potentially multi-attribute business identifier. However, we think in terms of the business identifier. The Time dimension created by the wizard creates an attribute called `CODE`, which it uses as the business identifier. It is a number that is used to represent the date for the level record. We will shortly use the **New Dimension Wizard** to create our Product dimension, and there we'll see how to specify a business identifier explicitly.

The format of the `CODE`, which is created automatically in a date dimension using the Time Wizard, is documented in the *OWB User's Guide* in Chapter 14 on *Defining Dimensional Objects* (`http://download.oracle.com/docs/cd/B28359_01/owb.111/b31278/toc.htm`). The section entitled *Using a Time Dimension in a Cube Mapping* explains that the format is YYYYMMDD where YYYY is the year, MM the two-digit month, and DD the two-digit day of the month. We'll need this information in Chapter 7 when we actually use the DATE_DIM dimension.

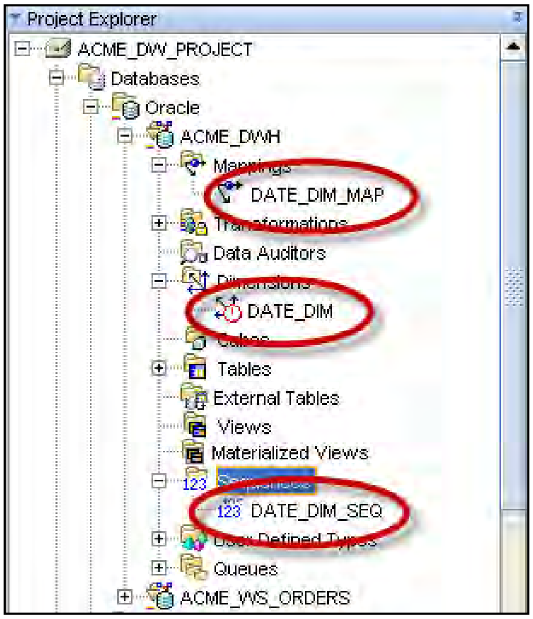
The **DATE_DIM_MAP** map entry that we can see in the previous image is a mapping for our `DATE_DIM` dimension, which can be run to populate the dimension. It will be created automatically for us by the wizard.

6. Continuing to the last step, it will display a progress bar as it performs each step and will display text in the main window indicating the step being performed. When it completes, we click on the **Next** button and it takes us to the final screen—the summary screen. This screen is a display of the objects it created and is similar to the previous display in step 5 of 6 that shows the pre-create settings. At this point, these objects have been created and we press the **Finish** button. Now we have a fully functional Time dimension for our data warehouse.

We could use the Data Object Editor to create our Time dimension, but we would have to manually specify each attribute, level, hierarchy, and sequence to use. Then we would have to create the mapping to populate it. So we definitely saved quite a bit of time by using the wizard.

Download at Boykma.Com

The **Time Dimension Wizard** does quite a bit for us. Not only does it create the Time dimension, but also creates a couple of additional objects needed to support it. Take a look at the following image, which is what our project explorer looks like after running this wizard:



Besides the dimension that it created, we now have a mapping that appears under the **Mappings** node. This is what we will deploy and run to actually build our Time dimension. We can also see that a sequence was created under the **Sequences** node. This is the sequence the dimension will use for the ID attribute that it created automatically as the surrogate identifier.

This completes our Time dimension, so let's look at the next dimension we're going to create. It is the dimension to hold the product information.

Download at Boykma.Com

The Product dimension

In the Product dimension, we will create the attributes that describe the products sold by ACME Toys and Gizmos. The principles of the Time dimension apply to this dimension as well. The same four characteristics need to be defined—Levels, Dimension Attributes, Level Attributes, and Hierarchies. The only difference will be that they are product-oriented instead of time/date-oriented.

Let's begin by looking at the attributes of our products, and then we'll group by levels and a hierarchy. The first thing we should consider is how each toy or gizmo sold by ACME is represented. As with any retail operation, a **Stock Keeping Unit** (**SKU**) is maintained that uniquely identifies each individual type of item sold. This is an individual number assigned by the main office that uniquely identifies each type of product sold by ACME, and there could be tens of thousands of different items. There could be more than one product with the same name, but they won't have the same SKU. So the SKU, together with the NAME, forms the business identifier we can use for the products. An SKU number all by itself is not very helpful. Therefore, in our Product dimension, we will want to make available more descriptive information about each product such as the description.

Every SKU can be grouped together by brand name—the toy manufacturer who makes the product—and then by the category of product, such as game, doll, action figure, sporting goods, and so on. Each category could be grouped by department in the store. Already, a list of attributes is starting to take shape and a product hierarchy is forming in our minds. For each of those levels in the hierarchy, that is the department, category, and brand, we need to have a business identifier. For that the NAME will be sufficient as there are no departments, categories, or brands that have the same name.

Let's put this down on paper to formalize it and add some more details.

Product Attributes (attribute type)

- ID (Dimension/Level)
- SKU (Level)
- Name (Dimension/Level)
- Description (Dimension/Level)
- List Price (Level)

Download at Boykma.Com

Product Levels

- Department located in
- Category of item
- Brand
- Item

Product Hierarchy (highest to lowest)

- Department
- Category
- Brand
- Item

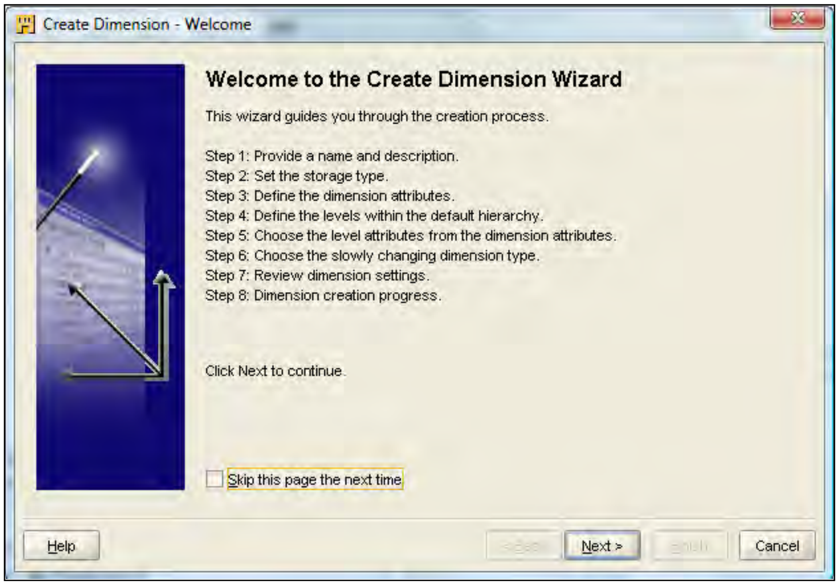
Looking at the product attributes, we see that they have been listed above with the type and that ID, Name, and Description are labeled as dimension attributes. This means they can appear on more than one level. Each level has a name (Item, Brand, Category, and Department) that identifies the level, but what about the names of the individual brands, or the different categories or departments? There has to be a place to store those names and descriptions, and that is the purpose of these dimension attributes. By labeling them as dimension attributes, they appear once for each level in the dimension. They are used to store the individual names and descriptions of the brands, categories, and departments. Likewise, each level will have a unique ID that will act as the surrogate key for that level, as well as one or more attributes defined as the business identifier. In our previous discussion about the Time dimension, we saw how a surrogate key was used as an identifier and how business identifiers were used; that same principle applies here.

As we want the computer to do most of the work for us, let's use the OWB Dimension Wizard to create our Product dimension now that we've determined what will be in it.

Download at Boykma.Com

Creating the Product dimension with the New Dimension Wizard

OWB provides a wizard that we can use to create a dimension. It is similar to the **Time Dimension Wizard** we used earlier, but is more generic for applying to other dimensions. As a result, there will be more steps involved in the wizard just because it has to ask us more because it will not be able to make as many assumptions as it did with the Time dimension. This wizard can be used with any dimension, and therefore things such as attributes, levels, and hierarchies are going to need to be defined explicitly. Right-click on the **Dimensions** node under our `ACME_DWH` **Oracle** module, which is under **Databases** in the **Design Center Project Explorer**. Choose **New** and then **Using Wizard...** to launch the Create Dimension Wizard. The very first screen we'll see is the **Welcome** screen that will describe for us the steps that we will be going through. We can see that it requires more steps than the Time Dimension Wizard:



Welcome to the Create Dimension Wizard

This wizard guides you through the creation process.

Step 1: Provide a name and description.
Step 2: Set the storage type.
Step 3: Define the dimension attributes.
Step 4: Define the levels within the default hierarchy.
Step 5: Choose the level attributes from the dimension attributes.
Step 6: Choose the slowly changing dimension type.
Step 7: Review dimension settings.
Step 8: Dimension creation progress.

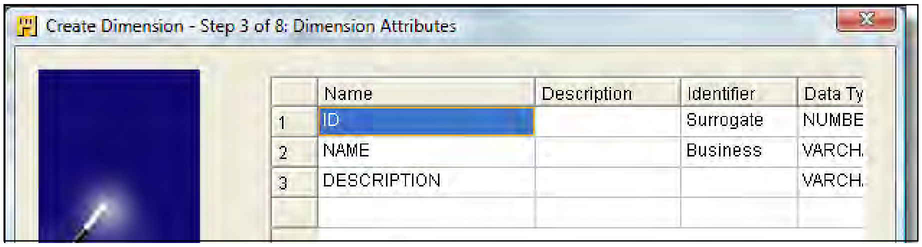
Click Next to continue.

Download at Boykma.Com

We will have to provide a name for our dimension, and tell it what type of storage to use—relational or multidimensional—just as we did for the Time Dimension Wizard. It will then ask us to define our dimension attributes. We didn't have to do that for the Time dimension. That wizard had a preset number of attributes it defined for us automatically because it knew it was creating a Time dimension. We then had to define the levels where we simply chose from a preset list of levels for the Time dimension. Here we have to explicitly name the levels. This is where we'll have to pay close attention to aggregations. We will then choose our level attributes from the dimension attributes.

Then we see in the previous figure that we will have to choose the **slowly changing dimension** type, which is how we want to handle changes to values in our dimension attributes over time. This is a new concept we haven't dealt with yet that pertains to dimensional modelling, and we'll soon briefly discuss just what that involves when we see the choices we'll be able to make for it. We'll then get a last chance to review the settings, and then it will create the dimension for us showing us the progress, which is similar to the last two steps of the Time Dimension Wizard.

1. After reviewing the steps, the wizard will go to the next screen where we enter a name for the dimension that we will call `Product`.
2. We'll then proceed to step 2, which is where we will select the ROLAP option for relational, as we did for the Time dimension.
3. Proceeding to step 3, we will be able to list the attributes that we want contained in our Product dimension. We see that the wizard was nice enough to create three attributes for us already—an **ID**, a **NAME**, and a **DESCRIPTION** as shown here:


Create Dimension - Step 3 of 8: Dimension Attributes

	Name	Description	Identifier	Data Ty
1	ID		Surrogate	NUMBE
2	NAME		Business	VARCH.
3	DESCRIPTION			VARCH.

Notice that the wizard has already labeled the **ID** as the **Surrogate Identifier** and the **Name** as the **Business Identifier**, and selected data types for those attributes for us. If we scroll that window to the right, we'll see that it has chosen sizes for the character attributes also. We can change all of these options at this point, so let's modify and add to this list to suit our Product dimension.

Download at Boykma.Com

As we enter the attributes and decide on sizes and types for them, we can look back in Chapter 2 where we defined our source data structure for the `ACME_POS` transactional database in SQL Server to get an idea of what types and sizes to use. We should make them at least as large as the source data so as not to lose any data when it gets loaded into the data warehouse.

We'll make the following changes:

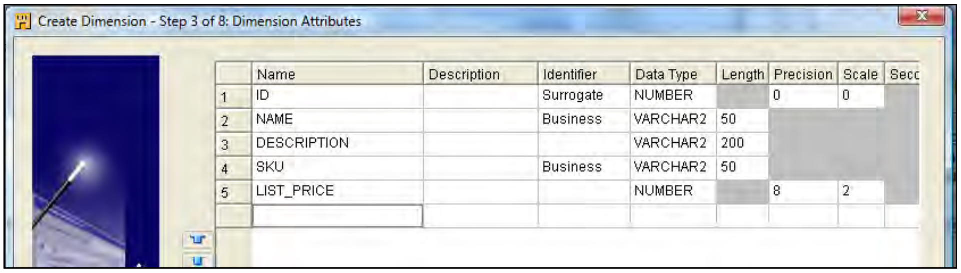
- Enter **SKU** in the name column on line 4 and leave the data type as **VARCHAR2**, but change the length to 50. Scroll the window to the right if any columns are not visible that need to be changed. We can also expand the dialog box to show additional columns.
- Enter **LIST_PRICE** in the name column on line 5, leave the data type as **NUMBER**, and leave the precision and scale as eight and two as it suggested.
- Make **SKU** a **Business Identifier** field in addition to **Name**. (Click on the drop-down box in the identifier column for **SKU**, and select **Business**.)
- Change the length of the **NAME** column from 25 to 50.
- Change the length of the **DESCRIPTION** column from 40 to 200.

Notice how the precision and scale were entered automatically for us by the Wizard when we entered names for our attributes. Moreover, they tended to make sense for the type of attribute. The **LIST_PRICE** had a default of eight for precision and two for the scale that we did not have to modify. If we choose logical names for our measures, it is able to make very good guesses as to what the precision and scale should be. **SKU** is a character field created with a varchar2 type with a reasonable length. Likewise, a **LIST_PRICE** amount implies money which requires a number having two decimal places (scale 2).

Suppose we make a mistake and enter a value and then decide not to keep it. Then we can delete the row by right-clicking on the row number to the left of the row, and then selecting **Delete** from the pop-up menu.

Download at Boykma.Com

The screen should now look like the following, expanded slightly to the right to see the additional length, precision, and scale columns:



	Name	Description	Identifier	Data Type	Length	Precision	Scale	Secc
1	ID		Surrogate	NUMBER		0	0	
2	NAME		Business	VARCHAR2	50			
3	DESCRIPTION			VARCHAR2	200			
4	SKU		Business	VARCHAR2	50			
5	LIST_PRICE			NUMBER		8	2	

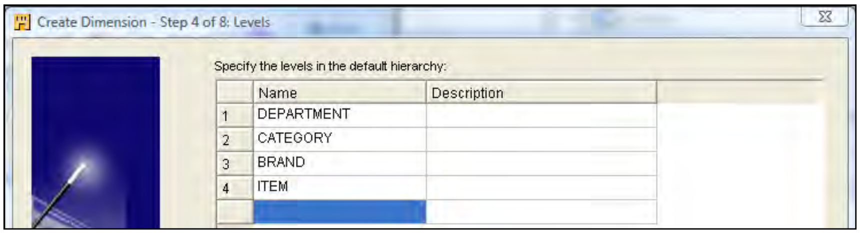
If we were to scroll that window all the way to the right, or expand it completely, we'd see even more columns such as the **Seconds Precision** and **Descriptor** column. If we press the **Help** button, it will explain what each column is. Briefly, the **Seconds Precision** is applicable to only TIMESTAMP data types, and expresses the precision of the seconds' portion of the value. The **Descriptor** is applicable to MOLAP (multidimensional) implementations and provides six standard descriptions that can be assigned to columns. It presets two columns, the `Long description` and the `Short description`. We can safely ignore them for our application.

4. The next step is where we can specify the levels in our dimension. There must be at least one level identified, but we are going to have four in our Product dimension. They are to be entered on this screen in order from top to bottom with the highest level listed first, then down to the lowest level. For our dimension, we'll enter **DEPARTMENT**, **CATEGORY**, **BRAND**, and **ITEM** in that order from top to bottom.

You might have noticed there is no step where we get to input hierarchies. The wizard will automatically create a default hierarchy called Standard that will contain the levels we enter here in this order. To create additional hierarchies, we must use the Data Object Editor after creating the dimension in the wizard.

Download at Boykma.Com

The dialog box should now look like this:



Create Dimension - Step 4 of 8: Levels

Specify the levels in the default hierarchy:

	Name	Description
1	DEPARTMENT	
2	CATEGORY	
3	BRAND	
4	ITEM	

5. Moving on to the next screen, we get to specify the level attributes. At the top are the levels, and at the bottom is the list of attributes with checkboxes beside each. If we click on each level in the top portion of the dialog box, we can see in the bottom portion that the wizard has preselected attributes for us. It chooses the three default attributes it created for us to be level attributes for each level, and the other two attributes—the `SKU` and `LIST_PRICE`—that we entered as level attributes for the bottom-most level—the `ITEM` level. We are not going to make any changes on this screen. The wizard has chosen wisely in this case. We could edit the descriptions of each of the level attributes if we wanted to.
6. This brings us to step 6 where we get to choose the **Slowly Changing Dimension (SCD)** type. This refers to the fact that dimension values will change over time. Although this doesn't happen often, they will change and hence the "slowly" designation. For example, we might have an `SKU` assigned to a Super Ball made by the ACME Toy Manufacturing Company, which then gets bought out by the Big Toy Manufacturing Company. This causes the Brand that is stored in the dimension for that `SKU` to change. In this screen we specify how we want to handle the change. We will have the following three choices, which are related to the issue of whether or how we want to maintain a history of that change in the dimension:
    - **Type 1**: Do not keep a history. This means we basically do not care what the old value was and just change it.
    - **Type 2**: Store the complete change history. This means we definitely care about keeping that change along with any change that has ever taken place in the dimension.
    - **Type 3**: Store only the previous value. This means we only care about seeing what the previous value might have been, but don't care what it was before that.

Download at Boykma.Com

The Type 2 and Type 3 options require additional licensing for our database if we want OWB to handle them automatically. We will need a license for the Warehouse Builder Enterprise ETL Option for that. As we are considering only basic functionality in this book, we'll leave this selection as Type 1 for now.

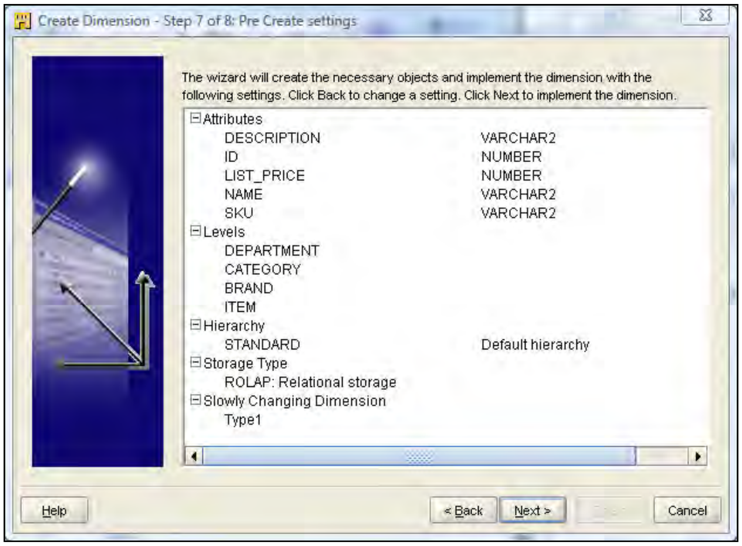
Handling SCDs can be done manually in a relational implementation. The Type 2 option to maintain a complete history would result in needing additional attributes where we want to maintain historical information. We need attributes designated as **Triggering attributes**. If changed, these attributes will generate a historical record. We also need an **Effective Date attribute** and an **Expiration Date attribute**. The Effective Date is when the record is entered. If a triggering attribute changes, the Expiration Date is set and a new record created with the updated information.

For the slowly changing Type 3 option, a new attribute in the dimension will be required to store only the most recent value.

With the Enterprise ETL option, the addition of these extra attributes and describing certain attributes as triggering attributes would be handled automatically for us.

The *Warehouse Builder User's Guide* documentation contains a more complete description of slowly changing dimensions. Also, there are other books available that cover dimensional modelling in depth, which give this topic much more coverage than we're able to provide here.

7. Moving on, we get our summary screen of the actions we performed. Here we can review our actions, and go back and make any changes if needed. It will look like the following, based on the selections we've made:

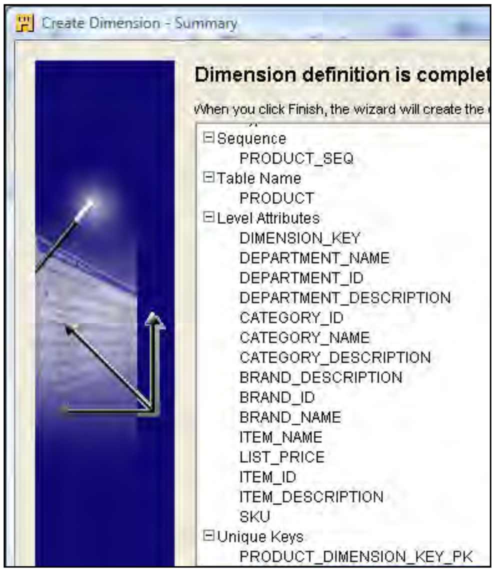


The wizard will create the necessary objects and implement the dimension with the following settings. Click Back to change a setting. Click Next to implement the dimension.

```
⊟Attributes
    DESCRIPTION                 VARCHAR2
    ID                          NUMBER
    LIST_PRICE                  NUMBER
    NAME                        VARCHAR2
    SKU                         VARCHAR2
⊟Levels
    DEPARTMENT
    CATEGORY
    BRAND
    ITEM
⊟Hierarchy
    STANDARD                    Default hierarchy
⊟Storage Type
    ROLAP: Relational storage
⊟Slowly Changing Dimension
    Type1
```

Download at Boykma.Com

8. Everything looks fine, so we move on to step 8. This step creates the dimension, showing us a progress bar as it does its work. It will report a successful completion when it's done, and clicking on the **Next** button at this point will bring us to the summary screen where we see the above information followed by additional information that the wizard has created for us based on our responses. The extra items are as shown here:



Create Dimension - Summary

Dimension definition is complet

When you click Finish, the wizard will create the

```
⊟Sequence
    PRODUCT_SEQ
⊟Table Name
    PRODUCT
⊟Level Attributes
    DIMENSION_KEY
    DEPARTMENT_NAME
    DEPARTMENT_ID
    DEPARTMENT_DESCRIPTION
    CATEGORY_ID
    CATEGORY_NAME
    CATEGORY_DESCRIPTION
    BRAND_DESCRIPTION
    BRAND_ID
    BRAND_NAME
    ITEM_NAME
    LIST_PRICE
    ITEM_ID
    ITEM_DESCRIPTION
    SKU
⊟Unique Keys
    PRODUCT_DIMENSION_KEY_PK
```

To reiterate, nothing has been physically created for us in the database yet. What the wizard has created for us are the definitions of our dimension, and the underlying table and other objects in OWB. The previous screen shows a **Sequence**, **Table Name**, and **Unique Key** that all correspond to objects that the wizard is creating for us in OWB. Later in Chapter 8, we'll deploy these objects to create the actual physical database objects in the target schema.

We can also see the additional attribute names that were added, which will become column names in the table to support the levels we identified. When we checked these boxes (or rather left checked the ones the wizard checked) beside the attributes for each level to indicate they were level attributes for that level back in step 5, it created additional names for each based on the level name.

Our Product dimension is now created and we can see it in the **Project Explorer** window under the **Dimensions** node under our `ACME_DWH` Oracle module.

Download at Boykma.Com

The Store dimension

We can create our Store dimension in a similar manner using the wizard. We will not go through it in much detail as it is very similar to how we created the Product dimension. The only difference is the type of information we're going to have in our Store dimension. This dimension provides the location information for our data warehouse, and so it will contain address information.

The creation of this dimension will be left as an exercise for the reader using the following details about the dimension.

Store Attributes (attribute type), data type and size, and (Identifier)

- ID (Dimension/Level): Leave default for type and size (Surrogate ID)
- Store_Number (Level, STORE only): VARCHAR2 length 10 (Business ID)
- Name (Dimension/Level): VARCHAR2 length 50 (Business ID)
- Description (Level, COUNTRY and REGION only): VARCHAR2 length 200
- Address1 (Level, STORE only): VARCHAR2 length 60
- Address2 (Level, STORE only): VARCHAR2 length 60
- City (Level, STORE only): VARCHAR2 length 50
- State (Level, STORE only): VARCHAR2 length 50
- ZipPostalCode (Level, STORE only): VARCHAR2 length 50
- County (Level, STORE only): VARCHAR2 length 255

Store Levels

- Country
- Region
- Store

Store Hierarchy (highest to lowest)

- Country
- Region
- Store

Download at Boykma.Com

Creating the Store dimension with the New Dimension Wizard

We will follow the same procedure as we had seen in the creation of the Product dimension. There are a few steps that are a little different from the previous procedure, and they are mentioned here.

In step 3, where we put in the attributes listed previously, we need to make sure not to forget to specify the surrogate and business identifiers. The surrogate identifier can stay as the default on the `ID`, but we will have to change the business identifier to be the `STORE_NUMBER`, which is a unique number that ACME Toys and Gizmos Company assigns to each of its stores.

[You may have the urge to include the region and/or country as an attribute in step 3, but resist the urge. They are being designated as levels. By specifying the level attributes to include the `Name` dimension attribute, we'll have our region and country included for us—as we'll see in a moment when we get to the final summary screen.]

In step 5 where we specify the level attributes, (the above-listed attributes that are applicable to each level) we need to specify all the attributes except `DESCRIPTION` for the `Store` level, and then just `ID`, `NAME`, and `DESCRIPTION` for the `Region` and `Country` levels. This is how we will include the region and country information.

[It may seem a bit redundant to include a description as well as a name for the `Country` and `Region` levels as our source data at the moment only includes one field to identify the country and region. However, this is needed to prevent an error from occurring later when we map data to this dimension. The same holds true for the Product dimension. If all we had were the `ID` and the `NAME`, those would be two key fields that cannot be changed for a record. There would be no descriptive information that could be changed, and the Warehouse Builder generates code for loading the dimension such that it requires at least one updatable field to be mapped without which the following error would occur:

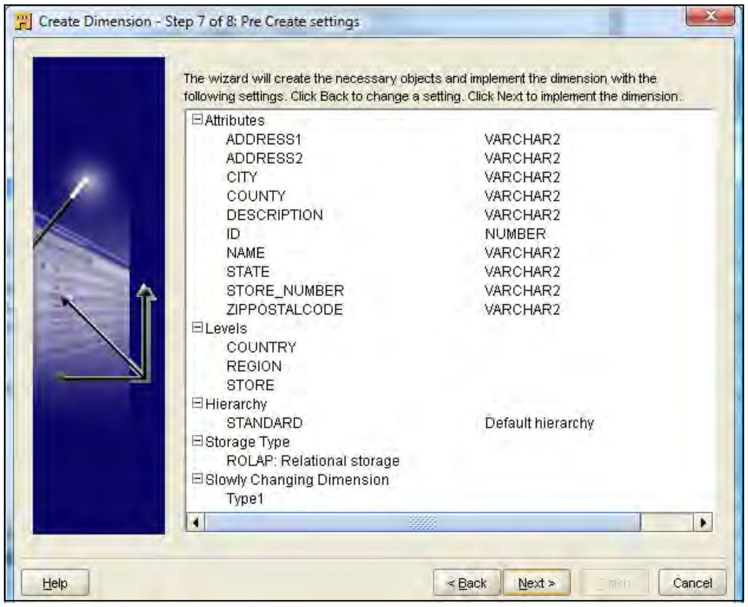
VLD-5005: No updatable inputs connected for dimension level <dimension><level>

At least one updatable input must be connected for level <dimension><level>, or the generated code will fail. Parent reference key and level natural key inputs are not updatable attributes in the target.

The New Dimension Wizard actually helps us to avoid this error by automatically including three attributes: an ID as the surrogate identifier, a NAME as the business identifier, and a DESCRIPTION as the updatable field.]

Download at Boykma.Com

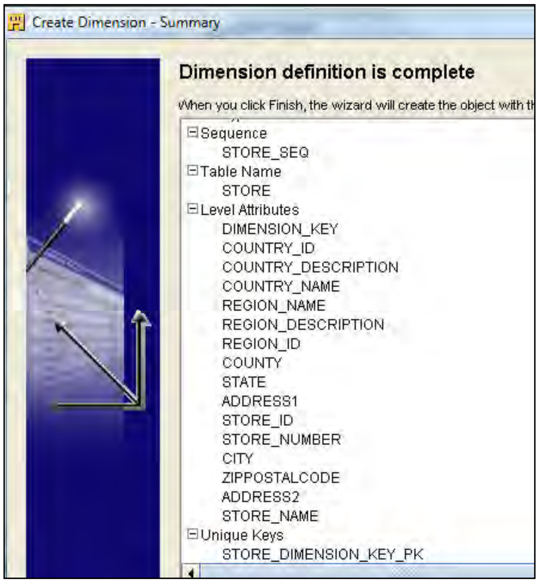
In step 7, the **Pre Create settings** page, as shown next, we can see what we should have specified for the Store dimension. We can click on the **Back** button to go back to make any changes.



The wizard will create the necessary objects and implement the dimension with the following settings. Click Back to change a setting. Click Next to implement the dimension.

```
⊟Attributes
    ADDRESS1              VARCHAR2
    ADDRESS2              VARCHAR2
    CITY                  VARCHAR2
    COUNTY                VARCHAR2
    DESCRIPTION           VARCHAR2
    ID                    NUMBER
    NAME                  VARCHAR2
    STATE                 VARCHAR2
    STORE_NUMBER          VARCHAR2
    ZIPPOSTALCODE         VARCHAR2
⊟Levels
    COUNTRY
    REGION
    STORE
⊟Hierarchy
    STANDARD              Default hierarchy
⊟Storage Type
    ROLAP: Relational storage
⊟Slowly Changing Dimension
    Type1
```

The final summary screen should look like the following when scrolled all the way to the bottom:



Dimension definition is complete

When you click Finish, the wizard will create the object with th

```
⊟Sequence
    STORE_SEQ
⊟Table Name
    STORE
⊟Level Attributes
    DIMENSION_KEY
    COUNTRY_ID
    COUNTRY_DESCRIPTION
    COUNTRY_NAME
    REGION_NAME
    REGION_DESCRIPTION
    REGION_ID
    COUNTY
    STATE
    ADDRESS1
    STORE_ID
    STORE_NUMBER
    CITY
    ZIPPOSTALCODE
    ADDRESS2
    STORE_NAME
⊟Unique Keys
    STORE_DIMENSION_KEY_PK
```

Download at Boykma.Com

Notice the region and country level attributes that are shown in the above image. This is where we see that information included as levels, instead of being specified as dimension attributes.

We'll make sure to save our work after creating this dimension, and then we'll move on to creating the cube.

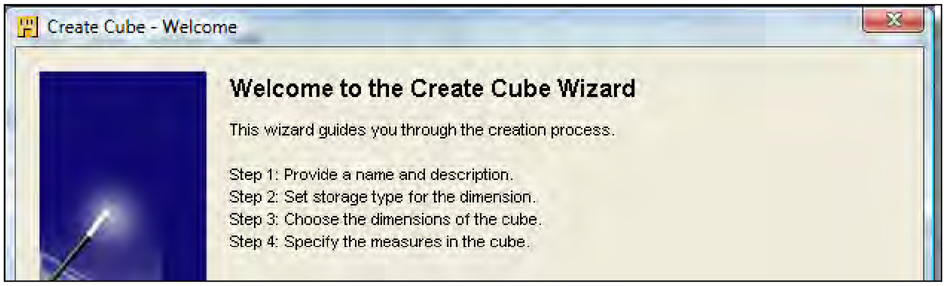
Creating a cube in OWB

Now that we have our dimensions defined, we have one last step to cover and our design for our data warehouse will be complete. We need to define our cube, which is where our measures will be stored—the facts that users will want to query. We discussed the design of our cube and agreed that we would store two measures, namely the sales amount and the number of items sold. We have already designed our three dimensions, and their links and measures will go together to make up the information stored in our cube.

There is a wizard available to us for creating a cube that we will make use of to ease our task. So let's start designing the cube with the wizard.

Creating a cube with the wizard

We will start the wizard in a similar manner to how we started up the Dimension wizard. Right-click on the **Cubes** node under the **ACME_DWH** module in **Project Explorer**, select **New**, and then **Using Wizard...** to launch the cube-creation wizard. The first screen will be the welcome screen, which will summarize the steps it will lead us through as shown in the following image of the main part of the welcome dialog box box:



Welcome to the Create Cube Wizard

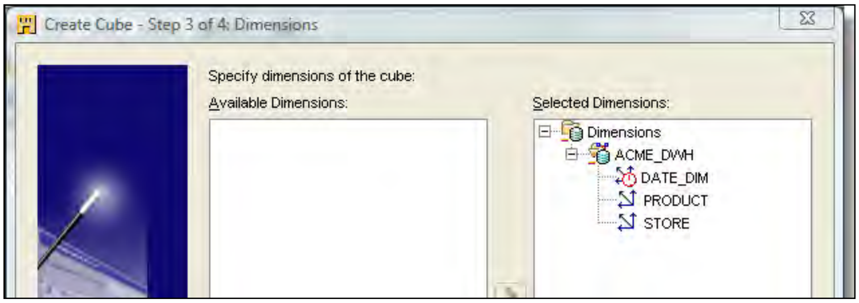
This wizard guides you through the creation process.

Step 1: Provide a name and description.
Step 2: Set storage type for the dimension.
Step 3: Choose the dimensions of the cube.
Step 4: Specify the measures in the cube.

Download at Boykma.Com

The following are the steps in the creation process:

1. We proceed right to the first step where we give our cube a name. As we will be primarily storing sales data, let's call our cube **SALES** and proceed to the next step.
2. In this step, we will select the storage type just as we did for the dimensions. We will select ROLAP for relational storage to match our dimension storage option, and then move to the next step.
3. In this step, we will choose the dimensions to include with our cube. We have defined three, and want all them all included. So, we can click on the double arrow in the center to move all the dimensions and select them. If we had more dimensions defined than we were going to include with this cube, we would click on each, and click on the single right arrow (to move each of them over); or we could select multiple dimensions at one time by holding down the *Ctrl* key as we clicked on each dimension. Then click the single right arrow to move those selected dimensions. This step looks like the following after we've made our selections:



Create Cube - Step 3 of 4: Dimensions

Specify dimensions of the cube:

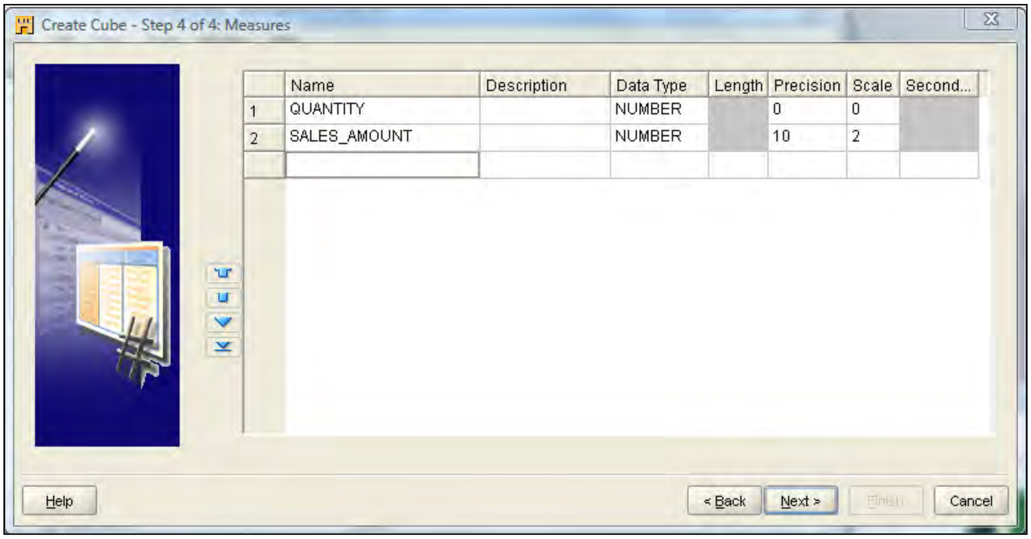
Available Dimensions:

Selected Dimensions:

- Dimensions
  - ACME_DWH
    - DATE_DIM
    - PRODUCT
    - STORE

4. Moving on to the last step, we will enter the measures we would like the cube to contain. When we enter **QUANTITY** for the first measure and **SALES_AMOUNT** for the second one, we end up with a screen that should look similar to this with the dialog box expanded to show all the columns:

Download at Boykma.Com



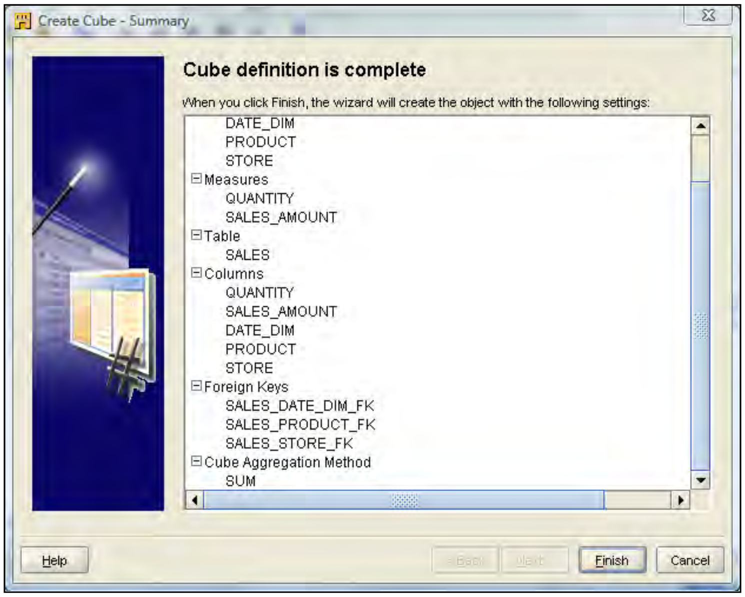
Create Cube - Step 4 of 4: Measures

	Name	Description	Data Type	Length	Precision	Scale	Second...
1	QUANTITY		NUMBER		0	0	
2	SALES_AMOUNT		NUMBER		10	2	

Help   < Back   Next >   Finish   Cancel

Clicking on **Next** in step 4 will bring us to the final screen where a summary of the actions it will take are listed. Selecting **Finish** on this screen will close the dialog box and place the cube in the **Project Explorer**.

The final screen looks like the following:



Create Cube - Summary

Cube definition is complete

When you click Finish, the wizard will create the object with the following settings:

```
    DATE_DIM
    PRODUCT
    STORE
⊟Measures
    QUANTITY
    SALES_AMOUNT
⊟Table
    SALES
⊟Columns
    QUANTITY
    SALES_AMOUNT
    DATE_DIM
    PRODUCT
    STORE
⊟Foreign Keys
    SALES_DATE_DIM_FK
    SALES_PRODUCT_FK
    SALES_STORE_FK
⊟Cube Aggregation Method
    SUM
```

Help   < Back   Next >   Finish   Cancel

Download at Boykma.Com

This dialog box works in a slightly different way than the dimension wizard. This final screen is the second-to-last screen when creating a dimension. The dimension wizard will present us with the progress screen as the final step. For cubes, the process is not quite as involved. That's because at this point, the cube is basically done with nothing left to do afterwards. So we may think we missed a step, but not to worry. Clicking on **Next** on this screen will exit the dialog box, and the cube will be created and will be accessible in the Project Explorer window.

Just as with the dimension wizard earlier, we get to see what the cube wizard is going to create for us in the Warehouse Builder. We gave it a name, selected the dimensions to include, and specified the measures. The rest of the information was included by the wizard on its own. The wizard shows us that it will be creating a table named `SALES` for us that will contain the referenced columns, which it figured out from the dimension and measures information we provided. At this point, nothing has actually been created in the database apart from the definitions of the objects in the Warehouse Builder workspace. We can verify that if we look under the **Tables** entry under our **ACME_DWH** database node. We'll see a table named `SALES` along with tables named `PRODUCT`, `STORE`, and `DATE_DIM`. These are the tables corresponding to our three dimensions and the cube.

You may have a slightly different table name. The wizard will not create a table with the same name as one already created, so it will append a unique number to the end to keep the table names from conflicting. This could happen if you've previously created a dimension with the same name, and then removed it and recreated it. It may not remove the associated table when you delete a cube or dimension object. The tables will appear in the **Project Explorer** under the **Tables** node. Expand that and you'll see the list of tables. Right-click a table and select **Delete**. The Warehouse Builder will ask if you really want to delete it, and will provide a checkbox to put the object in the recycle bin. Leave it checked just to be safe and click on **OK**, and the table will be removed.

The foreign keys we can see in the previous image are the pointers to the dimension tables. They will make the connection between our cube and our dimensions when they are deployed to the database.

Download at Boykma.Com

There is one final item that we did not specify and that is the cube aggregation method to be used. We saw earlier in the chapter how the multidimensional implementation contains behind-the-scenes functionality that we don't have to specify. Later we also saw how important it was to be aware of the aggregation of our measures, and whether they can be summed together at different levels and within the same level. The aggregation the cube will perform for us when we view different levels is one of those behind-the-scenes capabilities we would get with the OLAP feature.

When we view the region amounts, they will automatically be summed up from the amounts of the various stores in the region without us having to do anything extra. This is a nice feature the multidimensional implementation gives us, but aggregations are not created for the pure relational storage option. As we can generate either a relational or a multidimensional implementation, this had to be specified anyway and so it defaulted to sum. If we install the OLAP option or use a separate OLAP database in the future, we can change that aggregation method. But for now, we do not need it. It is possible to use aggregations with a pure relational implementation by creating separate summing tables, and there are OLAP data mining applications that can make use of them for more advanced implementations.

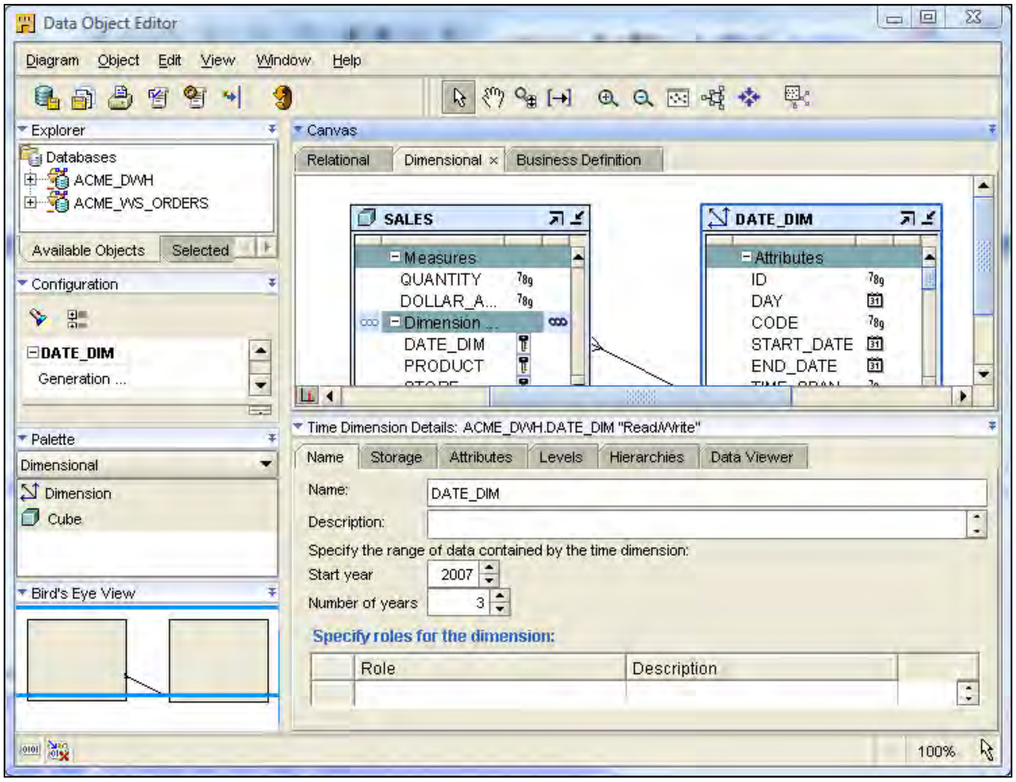
We click on the **Finish** button on this final screen and our sales cube is created. We'll save our work with the *Ctrl+S* key combination or from the design main menu. Our cube and dimensions are now complete. Let's take a look next at the Data Object Editor where we can view and edit our objects.

Using the Data Object Editor

We've mentioned the **Data Object Editor** previously. We used it in Chapter 2 to create our source metadata definitions for the `ACME_POS` transactional database, so let's take this opportunity to look a little closer at it. The Data Object Editor is the manual editor interface that the Warehouse Builder provides for us to create and edit objects. We did not have to use it to create a dimension, but more advanced implementations would definitely need to make use of it; for instance, to edit the cube to change the aggregation method that we just discussed. We'll take a brief look at it here before moving on to get an idea of some of the features it provides.

Download at Boykma.Com

We can get to the Data Object Editor from the **Project Explorer** by double-clicking on an object, or by highlighting an object (by selecting it with a single click), and then selecting **Edit | Open Editor** from the menu. Let's open the `DATE_DIM` dimension in the Data Object Editor and examine it as shown here:



Time Dimension Details: ACME_DWH.DATE_DIM "Read/Write"

Name: DATE_DIM

Description:

Specify the range of data contained by the time dimension:

Start year 2007

Number of years 3

Specify roles for the dimension:

Role	Description

All of the editors available to us in OWB have this same basic layout. Only the content of each section changes depending on what is being edited. We'll start in the upper right window and move around counter clockwise, discussing each window briefly.

- **Canvas**: Every editor has an area in which the contents are displayed graphically. This is called the Canvas. As we're in the Data Object Editor, the objects in the Canvas will be the objects that we created to hold data, which in this case are our cube and dimensions. Each object is displayed in a box with the name of the object as the title of the box and attributes of the object listed inside the box. These boxes can be moved around and resized manually to suit our tastes. There are three tabs available in the Data Object Editor Canvas: one for **Relational**, one for **Dimensional**, and one for **Business Definition**.

Download at Boykma.Com

They are for displaying objects of the corresponding type. As we're working with cubes and dimensions, these will be displayed on the **Dimensional** tab. If we were working with the underlying tables, they would have appeared on the **Relational** tab. The Business Definitions are for interfacing with the Oracle Discoverer Business Intelligence tool to analyze data.

We can see on our canvas that not only does the **DATE_DIM** dimension appear, but also a box for our **SALES** cube with a connecting line to the **DATE_DIM** dimension box. The Data Object Editor will display objects in context and this is showing us that the **DATE_DIM** object is referenced from the **SALES** Cube.

- **Explorer**: This is roughly analogous to the Project Explorer in the main Design Center interface, but it displays a subset of the objects that is applicable to the type of editor we have open. As this is the Data Object Editor, we can see other data objects in the **Explorer**. The **Available Objects** tab shows us objects that are available to include on our **Canvas**, the **Selected Objects** tab shows the objects that are actually currently on the **Canvas**, and will highlight the object currently selected.
- **Configuration:** The configuration window displays configuration information (properties) about items on our **Canvas**. If nothing shows in this window, just select an object in the Canvas by clicking on it and the configuration will appear. It is here that we can change the deployment option for the object to deploy OLAP metadata if we want a relational implementation to store the OLAP metadata. With the `DATE_DIM` dimension selected, scroll the Configuration window down to the **Identification** section, which contains a setting for the deployment option and we can see that it is set to deploy data objects only. For dimensions, the options are to deploy to catalog only (the OLAP catalog), deploy data objects only, or deploy all to do both. For cubes, there is an additional option to deploy aggregations.
- **Palette:** The Palette contains each of the objects that can be used in the Data Object Editor. In this case, they are all data objects. The list of objects available will change as the tab is changed in the Canvas to view different types of object. We can use this to create objects on our Canvas by clicking and dragging to the Canvas. This will create a new object where clicking and dragging from the Explorer will place an already created object on the canvas.
- **Bird's Eye View:** This window displays a miniature version of the entire Canvas and allows us to scroll around the Canvas without using the scroll bars. We can click and drag the blue-colored box around this window to view various portions of the main canvas, which will scroll as we move the blue box. We will find that in most editors, we will quickly outgrow the available space to display everything at once and will have to scroll around to see everything. This can come in very handy for rapidly scrolling the window.

Download at Boykma.Com

- **Dimension Details:** This is the window on the lower right and it contains details about the dimension we are currently editing. If nothing is displayed in the window, just click on the `DATE_DIM` dimension and its details will appear. Six tabs will appear, which display information for us. The names on those six tabs will change depending on the type of object we have selected. If we go ahead and click on the Sales cube, we'll see the tabs change. Back to the dimension, the tabs from left to right are as follows:
    - **Name:** This tab displays the name of the dimension along with some other information specific to the dimension type we are looking at. In this case, it's a Time dimension created by the Time Dimension Wizard and so it displays the range of data in our Time dimension.
    - **Storage**: Here we can see what storage option is set for our dimension object in the database, whether Relational or Multidimensional. If we wanted to switch between the two, this is where we could do it. For a relational implementation, we're able to specify a star or snowflake schema and whether we want to create **composite unique keys**. A composite key is one made up of more than one column to define uniqueness for a record. In most cases, it is a good idea to have this checked as it enforces uniqueness in the database for our dimension records. For a dimension, it will use the business identifiers we've specified as the key fields.
    - **Attributes**: The attributes tab is where we can see the attributes that are designed for our dimension. It displays the attributes in a tabular form allowing us to view and/or edit them, including adding new attributes or deleting the existing ones. It is here that we can also change the description of our attributes if we wanted, or add descriptions the wizard did not add.

You may have noticed by now that the attributes in our Time Dimension are not editable. They all appear as one solid background. We can scroll the window to display them and see what they are set to, but we can't change them. This is a feature of the Time dimension that was created by the wizard. It has created extra objects (as we saw earlier) to support the Time dimension like a mapping that could break if the wrong changes are made. So, it disallows changes. It is possible to modify the dimension behind the scenes to edit things, but that is a much more advanced topic.

Download at Boykma.Com

    - **Levels**: This is where we view and/or edit the levels for our dimension. We are able to edit some of the information on this tab for the Time dimension created by the wizard, but not all. We can check and uncheck boxes to indicate which of the various level types we want to use and which attributes are applicable to which level, but that is it. We are not able to add or remove any levels or attributes. If we were to view one of the other dimensions we created, it would be fully editable. For those other dimensions we could also assign different names and descriptions to the attributes for each level.
    - **Hierarchies**: This tab will let us specify hierarchy information for our dimension and will even let us create a new hierarchy. It's possible that we may have selected more levels on the previous page and now need to assign them to a hierarchy. There is also a **Create Map** button here that will automatically generate the mapping for us if we modify the hierarchies. This is one of the benefits of the Time dimension created by the wizard. Ordinary dimensions such as our Store and Product dimension will not have this **Create Map** button displayed on their **Hierarchies** tab.
    - **Data Viewer**: The Data Viewer is a more advanced feature that allows us to actually view the data in an object we are editing. This is only available for an object if it has been deployed to the database and has data loaded into it. It has a query capability to retrieve data and can specify a `WHERE` clause to get just the data we might need to see. For relational implementations, it will not display the data for a dimension or cube; but we can use it to view the data in the underlying table.
- **Cube Details:** If we click on the Sales Cube, the details window changes to display the details of our cube and the title changes to Cube Details. The tabs also change slightly:
    - **Name**: It has a name tab like the dimensions to display its name.
    - **Storage**: It has a storage tab as per dimensions. However, we see a different option here under the Relational (ROLAP) option where we can create **bitmap indexes**. An index is a database feature that allows faster access to data. It is somewhat analogous to the index of a book that allows us to get to a page in the book with the information we want much faster. A bitmap-type index refers to how it is stored in the database and is generally a better option to use for data warehouse implementations (so it is checked by default). There is also a composite unique key checkbox for cubes as there was for dimensions.

Download at Boykma.Com

For a cube, checking this box will create a unique key out of the foreign keys for the dimensions referenced by the cube. We want to check this box to ensure we can't enter duplicate data into our cube, that is, more than one cube record with the same set of dimension attributes assigned.

- **Dimensions**: Instead of attributes, the cube has a tab for dimensions. The dimensions referenced by a cube are basically its attributes.
- **Measures:** The next tab is for the measures of the cube. It is for those values that we are storing in our cube as the facts that we wish to track.
- **Aggregations**: Instead of hierarchies, a cube has aggregations. There are various methods of aggregation that we can select, as seen in the drop-down box, the most common of which is sum, which is the default. This is where the default aggregation method referred to earlier can be changed. There will be no aggregations in a pure relational implementation, so we will leave this tab set to the defaults and not bother changing it.
- **Data Viewer**: There is a tab for the data viewer to view cube data just as there is for a dimension. For pure relational implementations, it views the underlying table data.

These are the main features of the Data Object Editor. We can use it to view the objects the wizards have created for us, edit them, or create brand new objects from scratch. We can start with an empty canvas and drag new objects from the palette, or existing objects from the explorer, and then connect them. We will see other editors very similar to this from the next chapter when we start to look at ETL and mappings.

Summary

In this chapter we dove right in to creating our three dimensions and a cube using the Warehouse Builder Design Center. We used the Wizards available to help us out, as well as investigated the flexibility to manually create, view, and edit objects using the Data Object Editor. In a relatively short amount of time, we were able to design a data warehouse structure that could be used as is, or expanded to support more detailed information.

Now that we have our sources defined and our targets designed, it's time to start thinking about loading that target. Next, we'll look at some **Extract, Transform, and Load (ETL)** basics to lay the groundwork for designing the ETL we'll use to actually load data into our data warehouse.

Download at Boykma.Com

5 Extract, Transform, and Load Basics

We're moving along nicely into the process of designing and building a data warehouse. If you've been reading all the way through to here, you'll recall how we've introduced the Warehouse Builder software (how to install it along with the Oracle Database), looked at its architecture, and covered a short overview of the analysis and design phases for implementing a data warehouse project. We've defined our data sources and imported the metadata for them. We've designed our target structure into which we'll load the data. Congratulations for having read this far—don't give up now because we're not done yet. We still have to get data from our sources into our target. We will do that by:

1. Designing *mappings* in OWB.
2. Deploying the mappings to the database.
3. Running the mappings.

This chapter will expose **ETL** (**Extract, Transform, and Load**) for the first time in this book. ETL is the first step in building the mappings from source to target. We have sources and targets defined and now we need to:

1. Work on *extracting* the data from our sources.
2. Perform any *transformations* on that data (to clean it up or modify it).
3. *Load* it into our target data warehouse structure.

We will accomplish this by designing mappings in OWB. **Mappings** are visual representations of the flow of data from source to target and the operations that need to be performed on the data. However, before we can do that, we need to be familiar with what OWB offers us so that we can make best use of it.

Download at Boykma.Com

With this in mind, we'll spend this chapter looking at ETL in general and the Warehouse Builder features that support designing our ETL operations in particular. In the next chapter, we'll actually design our mappings in OWB.

ETL

The process of extracting, transforming, and loading data can appear rather complicated. We do have a special term to describe it, ETL, which contains the three steps mentioned. We're dealing with source data on different database systems from our target and a database from a vendor other than Oracle. Let's look from a high level at what is involved in getting that data from a source system to our target, and then take a look at whether to stage the data or not. We will then see how to automate that process in Warehouse Builder, which will relieve us of much of the work.

Manual ETL processes

First of all, we need to be able to get data out of that source system and move it over to the target system. We can't begin to do anything until that is accomplished, but what means can we use to do so? We know that the Oracle Database provides various methods to load data into it. There is an application that Oracle provides called **SQL*Loader**, which is a utility to load data from flat files. This could be one way to get data from our source system. Every database vendor provides some means of extracting data from their tables and saving it to flat files. We could copy the file over and then use the SQL*Loader utility to load the file. Reading the documentation that describes how to use that utility, we see that we have to define a control file to describe the loading process and definitions of the fields to be loaded. This seems like a lot of work, so let's see what other options we might have.

The Oracle Database allows us to create database links as we saw back in *Chapter 2*. When we define our sources, we can link them to other vendor's database systems via the heterogeneous services, which is exactly what we set up in *Chapter 2*. This looks like a better way to go. We could define a database link to point to our source database, and then we could directly copy the data into our database.

Download at Boykma.Com

However, our target database structure doesn't look anything like the source database structure. The POS Transactional database is a relational database that is highly normalized, and our target consists of cubes and dimensions implemented relationally in the database. How are we going to get the data copied into that structure? Clearly, there will be some manipulation of the data to get it reformatted and restructured from source to target. We cannot just take all the rows from one table in the source structure and copy them into a table in the target structure for each source table. The data will have to be manipulated when it is copied. This means we need to develop code that can perform this rather complex task, depending on the manipulations that need to be done.

In a nutshell, this is the process of extract, transform, and load. We have to:

1. *Extract* the data from the source system by some method.
2. Load flat files using SQL*Loader or via a direct database link. Then we have to *transform* that data with SQL or PL/SQL code in the database to match and fit it into the target structure.
3. Finally, we have to *load* it into the target structure.

The good news here is that the Warehouse Builder provides us the means to design this process graphically, and then generate all the code we need automatically so that we don't have to build all that code manually.

Staging

We need to consider a practical aspect to this process that is related to ETL, as well as to the structure in our target database. This practical aspect is the question of whether to stage the source data in a temporary location before performing the transformations on it and loading it into the target structure. **Staging** is the process of copying the source data temporarily into a table(s) in our target database. Here we can perform any transformations that are required before loading the source data into the final target tables. The source data could actually be copied to a table in another database that we create just for this purpose, but it doesn't have to be. This process involves saving data to storage at any step along the way to the final target structure, and it can incorporate a number of intermediate staging steps. The source and target designations will be affected during the intermediate steps of staging. So we'll need to decide on a staging strategy, if any, before designing the ETL in OWB. Now, we'll look at the staging process before we actually design any ETL logic.

Download at Boykma.Com

To stage or not to stage

There are a number of considerations we can take into account when deciding whether to use a staging area or not for our source data:

- The points to consider to *keep the process flowing as fast as possible* are:
    - The amount of source data we will be dealing with
    - The amount of manipulations of the source data that will be required
    - If the source data is in another database other than an Oracle Database, the reliability of the connection to the database and the performance of the link while pulling data across
- If a failure occurs during an intermediate step of the ETL process, we will have to restart the process. If such a failure occurs, we will have to consider the severity of the impact, as in the following cases:
    - Going back again to the source system to pull data if the first attempt failed.
    - The source data is changing while we are trying to load it into the warehouse, meaning that whatever data we pull the second time might be different from what we started with (and which caused the failure). This condition will make it difficult to debug the error that caused this failure.

These points will determine whether it makes sense to create a staging area. Suppose that we have a large amount of data to load and many transformations to perform on that data while loading. This process will take a lot longer if we directly access the remote database to pull and transform data, particularly if that remote database is not an Oracle Database. We'll also be doing all of the manipulations and transformations in memory and if anything fails, we'll have to start all over again. Any access to a remote database like this is going to have an impact on the performance of the database. Having to do it again just compounds the potential impact.

For example, in my current job, I am responsible for the process of loading data into our customer's warehouse and each ETL run pulls approximately 150,000 records. After transforming, these turn into 1,350,000 records in the target warehouse. There are several transformations done on the data. When directly accessing the remote database, which is not an Oracle Database, the process literally took hours during its first run. This is not an acceptable situation by any means. Through making various changes to configurations, including first adding a step to stage the data in the Oracle Database, the process was streamlined to take a total of about 25 to 30 minutes. That is acceptable for a data load that at most happens two or three times a week.

Download at Boykma.Com

The individual process to stage the data to a table in the Oracle database simply involves copying the data one-for-one over to the Oracle Database, and this runs in less than 30 seconds. This means the source database connection is only open for 30 seconds, whereas it had to constantly work for hours (previously) without a staging table. This is an example of how the source system is benefitted by using a staging table. An added benefit with the staging is that if the ETL process needs to be restarted, there is no need to go back to disturb the source system to retrieve the data.

Maybe there is only a small window of opportunity to grab the data at night when there is some downtime. Trying to perform all the transformations on the data at once while directly pulling it might cause the process to extend past the allowed time period. In this case, the above example of using a staging table in the Oracle Database will definitely be applicable. This will make the ETL process run very fast and the transformations can then be run on it without impacting the transactional system, or being impacted by it.

Configuration of a staging area

A staging area is clearly an advantage when designing our ETL. So we'll want to create one, but we will need to decide where we want to create it—in the database or outside the database. Outside the database, we would create a staging area in flat files on the file system that we could access to load data into the database. Back in *Chapter 2*, we discussed the pulling of data from flat files assuming that we weren't able to get access directly to the source database system and instead the DBA of ACME Toys and Gizmos company supplied us with a CSV file to import. For a staging area, we might want to consider storing the source data in a flat file ourselves, even if we have access to the source database. Our staging area in this case would be a folder on the file system and the data would be stored in a flat file.

When making a decision about whether to use a flat file, or create a table directly in the database, there are a number of particulars we'll want to take into account. As we are using OWB to design and implement our data warehouse, we've seen how it can directly support data stored in a flat file as a source (as if it was a table in the database) by using an external table. So this option of using OWB to directly support data stored in a flat file is open to us.

Download at Boykma.Com

The external tables in the Oracle Database now render some of the reasons for keeping source staging data in tables in the database moot. We can treat a flat file as essentially another table in the database. This option is available to us in the Warehouse Builder for defining a flat file as a source, but not a target. So we must keep this in mind if we want to stage data at some other point during the process after the initial load of data. Earlier, we needed to have all our data in database tables if we were relying solely on SQL in the database. External tables allow us to access flat files using all the benefits of SQL for querying the data, so now that reason is not as big a factor as it once was.

If you would like to read more about staging areas and flat files versus database tables, you can refer to the book titled "*The Data Warehouse ETL Toolkit – Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*", *Ralph Kimball and Joe Caserta, Wiley Publishing, Inc*. This book discusses a number of cases where it might be preferable to use flat files depending on the purpose at hand: things such as the storage and safekeeping of source data, sorting data, filtering it, or replacing text strings in data, and so on. Many of these purposes can be accomplished more efficiently outside the database with external tools and packages for manipulating text files. Unless you already have the packages or tools, it may make more sense to stick with database tables that can make full use of those features built into the database.

At the ACME Toys and Gizmos company, we are going to use a table in the database for the initial staging table. This way we can get some experience with how that would work in OWB using the Data Object Editor to create a table. One of our mapping tasks in the next chapter will be to create this table using OWB and use it as a part of our initial mapping from the source system. If needed, it is not difficult to switch to using a flat file later.

Mappings and operators in OWB

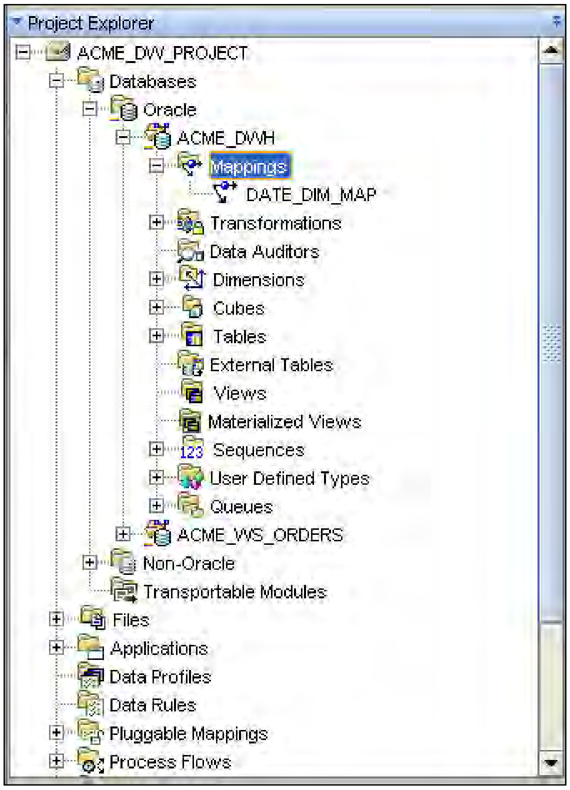
We are now going to look at the Warehouse Builder and its features for designing and building our ETL process. OWB handles this with what are called **mappings**. A mapping is composed of a series of **operators** that describe the sources, targets, and a series of operations that flow from source to target to load the data. It is all designed in a graphical manner using the **Mapping Editor**, which is available from the **Design Center**. Let's run the Design Center now and take a look at the Mapping Editor, its features, and some of the operators that are available to us. Launch the Design Center as we discussed in *Chapter 2* in the *Overview of Warehouse Builder Design Center* section.

Download at Boykma.Com

In the **Design Center | Project Explorer** window, expand the **ACME_DW_PROJECT** project (if it is not already expanded) by clicking on the plus sign beside it. To access the **Mapping Editor**, we need a mapping to work on. So to begin with, we can create an empty mapping at this point.

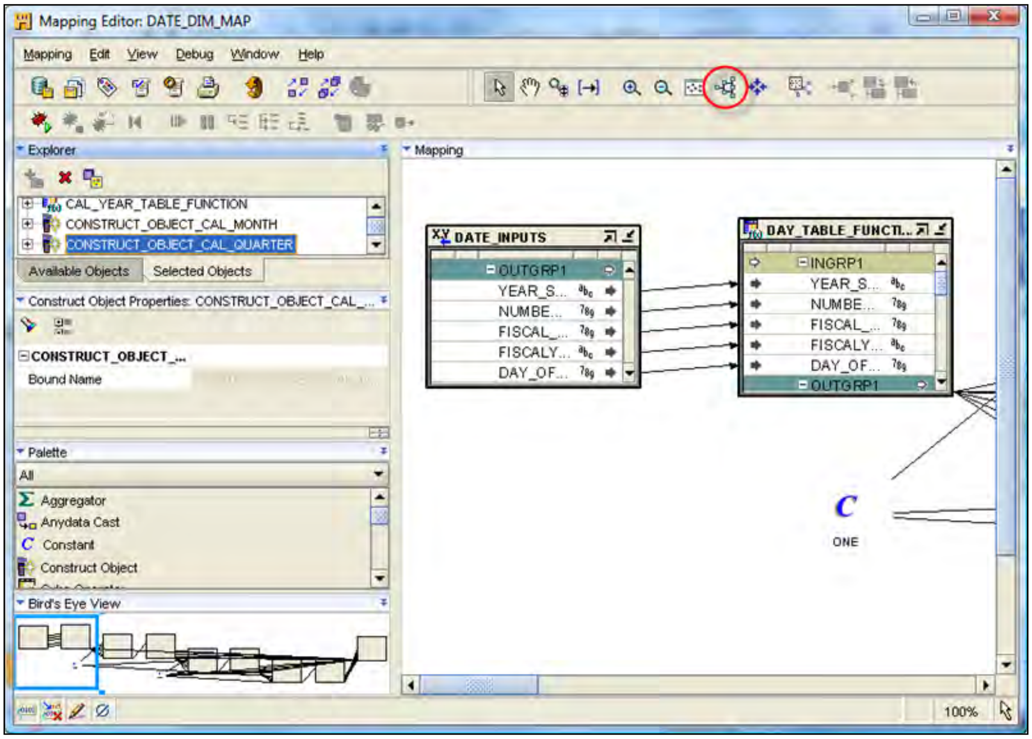
There is no wizard for creating mappings as there is for importing source metadata, or creating dimensions and cubes. There are too many options to lay out the mapping for a wizard, so it's difficult to make any kind of intelligent guesses as to how to design. However, there are some cases where we get a mapping for "free" such as the **DATE_DIM_MAP**, which was created for us automatically by the **Time Dimension** wizard—but that is the exception rather than the rule.

Mappings are created in the **Mappings** node. We can find it under the module we created to hold our data warehouse design under the **Databases | Oracle** node in our project. Expand that module, which we called **ACME_DWH**, and then expand the **Mappings** node underneath it. For reference, your **Project Explorer** should look like this now:



Download at Boykma.Com

The **DATE_DIM_MAP** we see under Mappings is the mapping that was created for us automatically by the **Time Dimension** wizard. Instead of creating a new mapping, which will have nothing in it yet, let's open this mapping and take a look at it for our initial exploration of the features in OWB for designing mappings. Let's double-click on the **DATE_DIM_MAP** mapping. It will launch the **Mapping Editor** and load the **DATE_DIM_MAP** into it. We are not going to modify it, but we will use the displayed **Mapping Editor** to familiarize ourselves with its features. This is very similar to the Data Object Editor for mappings, but not data objects. We're going to go into more detail in using the Mapping Editor simply because we need to use it to build our mappings; there is no wizard available to us. The **Mapping Editor** window looks like the following:



The **Mapping Editor** looks very similar to the **Data Object Editor** we saw in the last chapter, doesn't it? The window has been re-sized to better fit here, but we're going to maximize the window while we work with it for ease of use as we can view more information on the screen.

Download at Boykma.Com

Your view on the righthand side of the **Mapping Editor** might look like one single object instead of multiple objects with the connector lines drawn between them. Mappings created automatically like this one do not bother with any layout details, and just place all the objects in the same spot on the **canvas** (that big window on the right). We can go on clicking on the object and dragging it into a new location, thus revealing the one beneath it; but that's too much work. The **Mapping Editor**, as with all the editors, provides a convenient **Auto Layout** option that will do all that for us. Click on the **Auto Layout** button in the toolbar to spread everything out. It is circled (at the top) for your reference in the previous image.

Let's briefly discuss some of the main features we can see in the **Mapping Editor** screen. The *Oracle Warehouse Builder User's Guide*, which is available at `http://download.oracle.com/docs/cd/B28359_01/owb.111/b31278.pdf`, covers each of these windows in great detail. So we'll just touch upon them briefly, especially as we've seen some of them in the last chapter. We'll start with the main window on the right and then work down the left side. The windows of the **Mapping Editor** are:

- **Mapping**

  The **Mapping** window is the main working area on the right where we will design the mapping. This window is also referred to as the **canvas**. The **Data Object Editor** used the canvas to lay out the *data* objects and connect them, whereas this editor lays out the *mapping* objects and connects them. This is the graphical display that will show the operators being used and the connections between the operators that indicate the data flow from source to target.
- **Explorer**

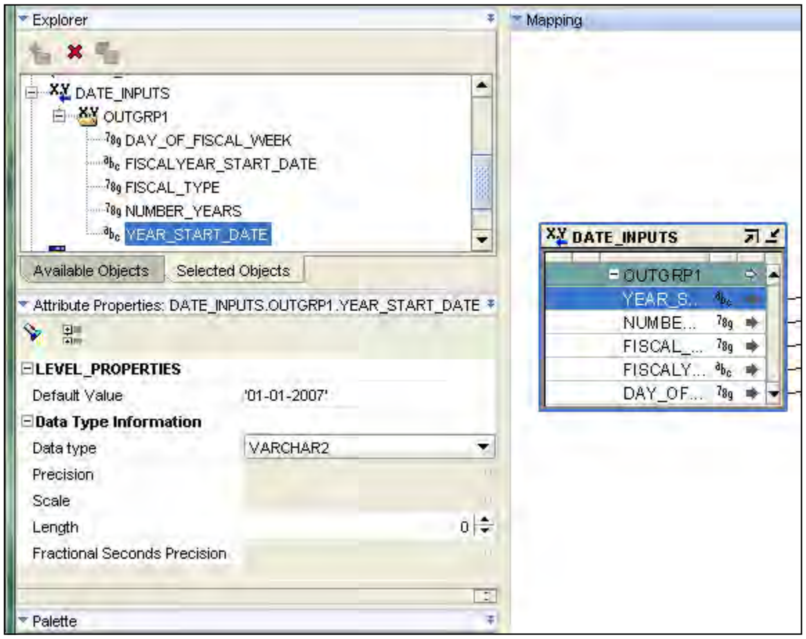
  This window is similar to the **Project Explorer** window from the Design Center, and is the same as the window in the **Data Object Editor**. It has the same two tabs—the **Available Objects** tab and the **Selected Objects** tab. It displays the same information that appears in the **Project Explorer**, and allows us to drag and drop objects directly into our mapping. The tabs in this window are:

  - **Available Objects**: This tab is almost like the **Project Explorer**. It displays objects defined in our project elsewhere, and they can be dragged and dropped into this mapping.
  - **Selected Objects**: This tab displays all the objects currently defined in our mapping. When an object is selected in the canvas, the **Selected Objects** window will scroll to that object and highlight it. Likewise, if we select an object in the **Selected Objects** tab, the main canvas will scroll so that the object is visible and we will select it in the canvas. Go ahead and click on a few objects to see what the tab does.

Download at Boykma.Com

- **Mapping properties**

  The mapping properties window displays the various properties that can be set for objects in our mapping. It was called the **Configuration** window in **Data Object Editor**. When an object is selected in the canvas to the right, its properties will display in this window. It's hard to see in the previous image, but there is a window at the bottom of the properties window that will display an explanation for the property currently selected. We can resize any of these windows by holding the mouse pointer over the edge of a window until it turns into a double arrow, and then clicking and dragging to resize the window so we can see the contents better. To investigate the properties window a little closely, let's select the **DATE_INPUTS** operator. We can scroll the **Explorer** window until we see the operator and then click on it, or we can scroll the main canvas until we see it and then click on the top portion of the frame to select it. It is the first object on the left and defines inputs into `DATE_DIM_MAP`. It is visible in the previous image. After clicking on it, all the properties will be displayed in the mapping properties window. But wait a minute, nothing is displayed for this object; not every object has properties. Let's click on one of the attributes—**YEAR_START_DATE**—within the **DATE_INPUTS** operator. It can be selected in the **Explorer** or canvas and is shown in the following image, which is a portion of the **Mapping Editor** window we're referring to. The windows have been resized to better display the information being referred to here:



Attribute Properties: DATE_INPUTS.OUTGRP1.YEAR_START_DATE

LEVEL_PROPERTIES	
Default Value	'01-01-2007'
Data Type Information	
Data type	VARCHAR2
Precision	
Scale	
Length	0
Fractional Seconds Precision	

Download at Boykma.Com

Now we can see some properties. **YEAR_START_DATE** is an attribute of the **DATE_INPUTS** object and defines the starting date to use for the data that will be loaded by this mapping. The properties that can be set or displayed for it include the characteristics about this attribute such as what kind of data type it is, its size, and its default value. Recalling our running of the **Time Dimension Wizard** in the last chapter, there was one option to specify the year range for the dimension and we chose **2007** as the start year and that is what formed the source for the default value we can see here. Do not change anything but just click on a few more objects or attributes to look around at the various properties.

- **Palette**

  The **Palette** contains each of the objects that can be used in our mapping. We can click on the object we want to place in the mapping and drag it onto the canvas. This list will be customized based on the type of editor we're using. The **Mapping Editor** will display mapping objects. The **Data Object Editor** will display data objects.

- **Bird's Eye View**

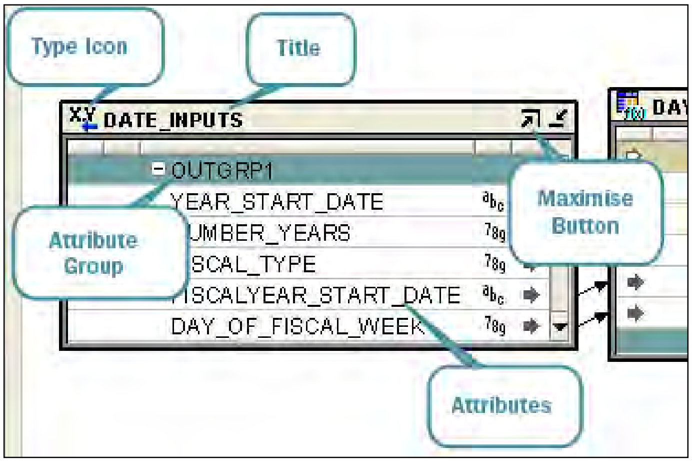
  This window displays a miniature version of the entire canvas and allows us to scroll around the canvas without using the scroll bars. We can click and drag the blue-colored box around this window to view various portions of the main canvas. The main canvas will scroll as we move the blue box. Go ahead and give that a try. We will find that in most mappings, we'll quickly outgrow the available space to display everything at once and will have to scroll around to see everything. This can come in very handy for rapidly scrolling the window.

The canvas layout

Let's take a closer look at some of the general features of the operators we can see in our canvas, and then at some of the features specific to different operators. Much of what we'll cover here is also applicable to the **Data Object Editor**. Operators on the canvas are represented by boxes with a title that indicates the name of the operator and an icon that indicates the type. In the canvas, we'll take a look at the operator that is on the far left of the canvas called **DATE_INPUTS**. This operator happens to be a **Mapping Input Parameter** operator.

Download at Boykma.Com

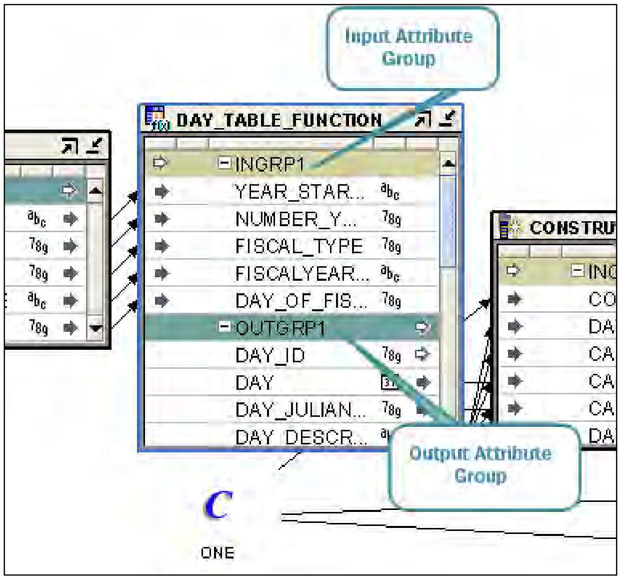
It is shown in the following screenshot with the key features highlighted with callouts:



The box can be resized by clicking the left mouse button and holding it over an edge of the operator, and then dragging it to a new size. To show the entire contents at once, you can click on the maximize button to instantly expand the box, as shown in the above screenshot. We can see some lines of information listed inside the box, which are the attributes of this operator. There are two major types of attributes—an input group and an output group. In this case, we can see one group named **OUTGRP1** which tells us this operator has only an output group. This operator represents the input of information into the map at the beginning, so it is not meant to have any other operators mapped as input to it. Thus, it has no input attributes, but only output attributes.

Take a look at the operator on the right of the **DATE_INPUTS** operator called **DAY_TABLE_FUNCTION**. It has both input and output attributes as shown in the next screenshot, because this operator represents a PL/SQL function. A PL/SQL function takes the values supplied as input attributes in the `INGRP1` group as parameters to the function and returns the value indicated in the `OUTGRP1` group as a return value from the function.

Download at Boykma.Com



Renaming attribute group names

The names **INGRP1** and **OUTGRP1** are generic names that OWB uses as default names for input and output groups when there is only one of each. We can change those names if we want to, but will find that in general the default names are fine when there is only one input and one output group.

There are some operators that contain more than one input or output group. For instance, **Joiner Operator** is an operator to represent a join of two or more tables. This operator will have an input group defined for each table. In that case, each input group would have a different number incrementing from `1` (for example, `INGRP2`, `INGRP3`, and so on). It might make sense in this case to rename the input groups to match the tables being joined, but this is not required.

An attribute group name is edited in the **Details** window for the group name. This window is accessible by right-clicking on the group name in the canvas and selecting **Open Details...** from the pop-up menu. We'll be making use of that **Details** window in the next chapter when we actually start building a mapping.

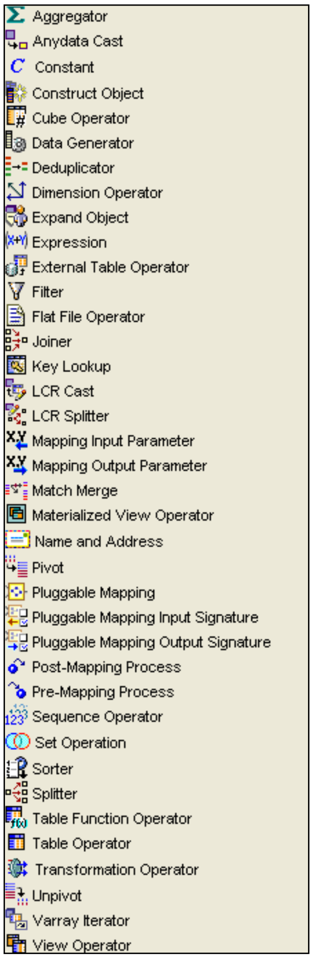
We can also note with `DAY_TABLE_FUNCTION` that the number of attributes is greater than what's displayed in the operator window. We can scroll down through the list of attributes, or resize the operator in the canvas to see more of them.

Download at Boykma.Com

That was a brief introduction to the user interface for the Mapping Editor and the operators as displayed in the canvas. The *Oracle Warehouse Builder User's Guide* includes additional details about the various windows and their purpose, as well as the toolbars and menus that are available. We will now look at the various operators that OWB provides to us in a little more detail.

OWB operators

We'll discuss here the various operators using the same category breakdown that the *Oracle Warehouse Builder User's Guide* uses in its section on the types of operators—**Source and Target Operators**, **Data Flow Operators**, and **Pre/Post Processing Operators**. All of the operators are available to us from the **Palette** window in the **Mapping Editor**, so we can refer to it as we discuss each operator. The following screenshot displays the complete list of operators:



Aggregator
Anydata Cast
Constant
Construct Object
Cube Operator
Data Generator
Deduplicator
Dimension Operator
Expand Object
Expression
External Table Operator
Filter
Flat File Operator
Joiner
Key Lookup
LCR Cast
LCR Splitter
Mapping Input Parameter
Mapping Output Parameter
Match Merge
Materialized View Operator
Name and Address
Pivot
Pluggable Mapping
Pluggable Mapping Input Signature
Pluggable Mapping Output Signature
Post-Mapping Process
Pre-Mapping Process
Sequence Operator
Set Operation
Sorter
Splitter
Table Function Operator
Table Operator
Transformation Operator
Unpivot
Varray Iterator
View Operator

Download at Boykma.Com

Earlier in the *ETL* section of this chapter, we discussed the various means at our disposal for performing ETL operations manually with applications such as SQL*Loader. We have also mentioned that OWB allows us to define the process graphically, and then generates the code for us as well. The bottom line is that OWB makes use of the existing facilities within the database and the utilities supplied with the database to accomplish the data load and transformations.

[The operators we use will determine the kind of code that gets generated by OWB. Keep this in mind as we study these operators because it will help us understand some of the explanations that are supplied for the operators in the documentation.]

Most of the operators will result in a PL/SQL mapping. So the explanations are in terms of the SQL or PL/SQL code element that is created for an operator.

As we can see, we have quite an extensive list of operators available to us and we won't have room here to talk about all of them. We won't need them all, and that is usually the case when designing mappings in OWB. We will be focusing on the main ones that we will need for our application and discuss some of the more common ones along with the ones we can see in the **DATE_DIM_MAP**. The operators found in the **DATE_DIM_MAP** will be pointed out, so you can see an example by looking at that map. We'll talk about some of the operators in more detail as we actually begin to use them in the next chapter. As much as possible, we will try to discuss the operators from a functionality standpoint without getting too bogged down by the actual code that is generated.

[For the adventurous out there, you can take a look at the code that the Warehouse Builder will create for the mapping in the database. But unless you are an SQL coding wizard, you will become quickly overwhelmed. There is OWB-generated code at my current job that contains SQL insert statements that are over 650-lines long for a single statement. This is definitely not for the faint-hearted. Have no fear, however; we don't have to dig into the code if we don't want to. This is the beauty of what a graphical interface does for us. For those who do love to delve into the code, there are ways to view it but that is definitely a more advanced topic.]

Download at Boykma.Com

Source and target operators

The Warehouse Builder provides operators that we will use to represent the sources of our data and the targets into which we will load data. We know we're going to be pulling data from non-Oracle database tables, and loading it into dimensions and cubes in our Oracle Database. We also saw in *Chapter 2* how to import metadata from a flat file source. So there is another source type that we'll need to handle, a flat file. With that in mind, here are some of the operators we're going to potentially need:

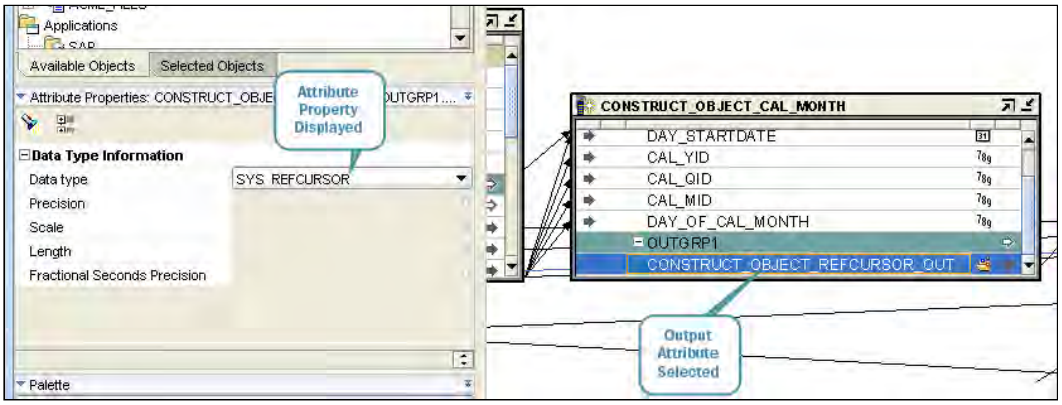
- **Cube Operator**—an operator that represents a cube that we have previously defined. We defined our cube back in *Chapter 4* and this operator will be used to represent that cube in our mapping.
- **Dimension Operator**—an operator that represents previously defined dimensions. As with our cube, our dimensions were defined in *Chapter 4* and this operator will be used in our mapping to represent them. We can see an example of **Dimension Operator** in `DATE_DIM_MAP`. This mapping is designed to load our `DATE_DIM` dimension, and so an operator of the same name was created in it at the end on the far right of the canvas.
- **External Table Operator**—this operator represents external tables, which we have seen in *Chapter 2*. They can be used to access data stored in flat files as if they were tables. We will look at using an external table to access the flat file that we imported back in *Chapter 2*.
- **Table Operator**—this operator represents a table in the database. We will need to store data in tables in our Oracle Database at some point in the loading of data.

Those are the main operators we're going to need. There are a number of other operators that are defined for use as sources and targets in our mappings that can be very useful. The following are some of the more common operators:

- **Constant**—represents a constant value that is needed. It can be used to load a default value for a field that doesn't have any input from another source, for instance. The `DATE_DIM_MAP` mapping contains a couple of constant values to represent hardcoded numbers. One is named `ONE` for the number 1, and one is named `ZERO` for a 0.
- **View Operator**—represents a database view. Source data is frequently retrieved via a view in the source database that can pull data from multiple sources into a single, easily accessible view.
- **Sequence Operator**—can be used to represent a database **sequence**, which is an automatic generator of sequential unique numbers and is most often used for populating a primary key field.

Download at Boykma.Com

- **Construct Object**—this operator can be used to actually construct an object in our mapping. There are examples of this in `DATE_DIM_MAP`, which builds the `DATE_DIM` dimension. An *object* in this context refers to a PL/SQL object. We can see three **Construct Object** operators in `DATE_DIM_MAP`—for a calendar month (`CONSTRUCT_OBJECT_CAL_MONTH`), a calendar quarter (`CONSTRUCT_OBJECT_CAL_QUARTER`), and a calendar year object (`CONSTRUCT_OBJECT_CAL_YEAR`). If we click on the attribute in the **OUTGRP1** of one of those construct operators, we can see in the **Attribute Properties** window on the left that it is of type **SYS_REFCURSOR**. An example is shown in the next screenshot with the **CONSTRUCT_OBJECT_REFCURSOR_OUT** attribute selected in the **CONSTRUCT_OBJECT_CAL_MONTH** object.



A **SYS_REFCURSOR** is a PL/SQL type that represents a **cursor** in PL/SQL. A cursor is used to point to the row of the result of the query that is defined for that cursor. This is a rather advanced topic to be covered in this book, but is mentioned here as **DATE_DIM_MAP** contains some of this type.

These all represent sources and targets of data for our mappings. When we drag and drop one of these operators onto our canvas, it represents an actual database object. When created, every one of these will need to be bound to its underlying database object as we are using the relational storage option. For tables, attributes of the table operator will correspond to columns in the table, likewise for views and external tables. The same principle holds true for the cube and dimension operators. Attributes of the operator correspond to the attributes of the dimension or cube. If the dimension or cube is implemented relationally, they will correspond to the columns of the underlying table that is created.

Download at Boykma.Com

A constant is implemented in the database as PL/SQL code using the PL/SQL syntax for representing a constant value. The value will be the value we would set on a constant's attribute. For sequences, constants are implemented as a database **sequence** object. The code is generated to invoke constants to retrieve the `currval` or `nextval` value from the underlying sequence as needed, depending on how we used it in our mapping. The `currval` variable will return the current value of the sequence, and the `nextval` variable will return the next value.

Data flow operators

Sources and targets are good and we could end right there by connecting our sources directly to our targets. This would result in a complete mapping that would load our target from our source, but this would mean there needs to be a one-to-one correspondence between our source and target. If that were the case, why bother creating a data warehouse target in the first place if it's only going to look exactly like the source? Just query the source data.

The true power of a data warehouse lies in the restructuring of the source data into a format that greatly facilitates the querying of large amounts of data over different time periods. For this, we need to transform the source data into a new structure. That is the purpose of the **data flow operators**. They are dragged and dropped into our mapping between our sources and targets. Then they are connected to those sources and targets to indicate the flow of data and the transformations that will occur on that data as it is being pulled from the source and loaded into the target structure. Some of the common data flow operators we'll see are as follows:

- **Aggregator**—there are times when source data is at a finer level of detail than we need. So we need to sum the data up to a higher level, or apply some other aggregation type function such as an average function. This is the purpose of the **Aggregator** operator. This is implemented behind the scenes using an SQL `group by` clause with an aggregation SQL function applied to the amount(s) we want to aggregate.
- **Deduplicator**—sometimes our data records will contain duplicate combinations that we want to weed out so we're loading only unique combinations of data. The **Deduplicator** operator will do this for us. It's implemented behind the scenes with the `distinct` SQL function, which returns combinations of data elements that are unique.

Download at Boykma.Com

- **Expression—**this represents an SQL expression that can be applied to the output to produce the desired result. Any valid SQL code for an expression can be used, and we can reference input attributes to include them as well as functions.

It's possible to write expressions in an **Expression** operator for which separate operators are predefined such as functions. We will generally get better performance out of our mappings if we use the prebuilt operators whenever possible rather than implement code in expressions. So, if there is an operator available, we'll use it and use an expression only if we have to.

- **Filter—**this will limit the rows from an output set to criteria that we specify. It is generally implemented in a `where` clause in SQL to restrict the rows that are returned. We can connect a filter to a source object, specify the filter criteria, and get only those records that we want in the output.
- **Joiner—**this operator will implement an SQL `join` on two or more input sets of data. A `join` takes records from one source and combines them with the records from another source using some combination of values that are common between the two. We will specify these common records as an attribute of the `join`. This is a convenient way to combine data from multiple input sources into one.
- **Key Lookup—**a **Key Lookup** operator looks up data in a table based on some input criteria (the key) to return some information required by our mapping. It is similar to a **Table Operator** that was discussed previously for sources and targets. However, a **Key Lookup** operator is geared toward returning a subset of rows from a table based on the key criteria we specify, rather than representing all the rows of a table, which the **Table Operator** does. It can look up data from a table, view, cube, or dimension.
- **Pivot—**this operator can be useful if we have source records that contain multiple columns of data that is spread across columns instead of rows. For instance, we might have source records of sales data for the year that contain a column for each month of the year. But we need to save that information by month, and not by year. The **Pivot** operator will create separate rows of output for each of those columns of input.

Download at Boykma.Com

- **Set Operation**—this operator will allow us to perform an SQL set operation on our data such as a `union` (returning all rows from each of two sources, either ignoring the duplicates or including the duplicates) or `intersect` (which will return common rows from two sources).
- **Splitter**—this operator is the opposite of the **Joiner** operator. It will allow us to split an input stream of data rows into two separate targets based on the criteria we specify. It can be useful for shunting rows of data off to a side error table to flag them while copying the good rows into the main target.
- **Transformation Operator**—this operator can be used to invoke a PL/SQL function or procedure with some of our source data as input to provide a transformation of data. For instance, the SQL `trim()` function can be represented by **Transformation Operator** to take a column value as input, and provide the value as output after having any whitespace trimmed from the value. This is just one example of a function that can be implemented with the **Transformation Operator**. There are numerous others available to us.

A **Transformation Operator** is an example of an operator that could be implemented in an **Expression** operator by simply invoking the `trim()` SQL function directly on an input value. But as we can implement a `trim()` directly using its own operator, we should do so for efficiency and consistency.

- **Table Function Operator**—a **Table Function Operator** can be seen in the `DATE_DIM_MAP` map. There are three **Table Function** operators defined: `CAL_MONTH_TABLE_FUNCTION`, `CAL_QUARTER_TABLE_FUNCTION`, and `CAL_YEAR_TABLE_FUNCTION`. This kind of operator represents a **Table Function**, which is defined in PL/SQL and is a function that can be queried like a table to return rows of information. The **Table Function Operators** are more advanced than we will be covering in this book, but are mentioned here as `DATE_DIM_MAP` includes them.

These are just some of the operators available to us for performing transformations on our data as it flows from source to target. Others are described in the *Oracle Warehouse Builder User's Guide* (`http://download.oracle.com/docs/cd/B28359_01/owb.111/b31278.pdf`).

Download at Boykma.Com

Pre/post-processing operators

There is a small group of operators that allow us to perform operations before the mapping process begins, or after the mapping process ends. These are the **pre-** and **post-processing operators**. We can perform functions or procedures before or after a mapping runs, and can also accept input or provide output from a mapping process.

- **Mapping Input Parameter**—this operator allows us to pass a parameter(s) into a mapping process. It is very useful to make a mapping more generic by accepting a constant value as input that might change, rather than hardcoding it into the mapping. `DATE_DIM_MAP` uses a **Mapping Input Parameter** operator as its very first operator on the left, which we discussed earlier when talking about Mapping Properties.
- **Mapping Output Parameter**—as the name suggests, this is similar to the **Mapping Input Parameter** operator; but provides a value as output from our mapping.
- **Post-Mapping Process**—allows us to invoke a function or procedure after the mapping completes its processing. There may be some cleanup we want to do automatically such as deleting all the records from a table we're done with—perhaps a staging table that was used during the mapping process.
- **Pre-Mapping Process**—it's not too hard to figure out what this operator does. It allows us to invoke a function or procedure before the mapping process begins. Maybe our mapping needs to do a key lookup of a data value that is going to be stored in every row of output. But we don't want to invoke a **Key Lookup** operator for every record of input. So we could use a **Pre-Mapping Process** operator instead to invoke the function once at the beginning, which will make the returned value available for every row that is processed without having to re-invoke the procedure.

If you are looking at the *Oracle Warehouse Builder User's Guide* section that discusses these categories of operators, you no doubt noticed a fourth category called **Pluggable Mappings**. This is a more advanced feature we won't cover in this introductory book on OWB. It allows us to create a grouping of operators that can function as a single operator and be reused in other mappings.

Download at Boykma.Com

Summary

This chapter has given us an overview of the Extract, Transform, and Load (ETL) process as well as the Warehouse Builder's support for designing our ETL process. We discussed the process of mapping and a little of what that involves in OWB. We took a look at the OWB **Mapping Editor** to get a feel for the windows available to us, and also looked at a list of some of the operators OWB provides for us to use in our mappings.

We're laying the groundwork here for the real fun that comes in the next chapter where we get to put this knowledge to use in designing a mapping. In the next chapter, we will also get to use some of these operators.

Download at Boykma.Com

6
ETL: Putting it Together

We had our first introduction to the process of ETL in the last chapter where we discussed what it is and saw the features the Warehouse Builder has for designing our ETL processes. We looked at the Mapping Editor, which is the main interface we'll use to build our ETL mappings. We also looked at the objects in OWB that we can use. However, we didn't get to do anything other than just look. We have all this new knowledge and are ready to use it. So let's work on designing a mapping, which will make use of some of the features we looked at in the last chapter.

We've looked in detail at the source structures in *Chapter 2* and talked about staging data in *Chapter 5*. In the previous chapters, we've also talked about the concept of extraction, transformation, and loading of data that will be required to get the source data from our source to our target structure. We will get to put all this together in this chapter as we begin to design and build our mappings. You may have already started thinking of some ideas to handle the mapping of information into our target, or some issues that we'll need to address. So without further ado, let's get started.

Designing and building an ETL mapping

We are going to design and build our very first ETL mapping in OWB, but where do we get started? We know we have to pull data from the `ACME_POS` transactional database as we saw back in *Chapter 2*. The source data structure in that database is a normalized relational structure, and our target is a dimensional model of a cube and dimensions. This looks like quite a bit of transforming we'll need to do to get the data from our source into our target. We're going to break this down into much smaller chunks, so the process will be easier.

Download at Boykma.Com

Instead of doing it all at once, we're going to bite off manageable chunks to work on a bit at a time. We will start with the initial extraction of data from the source database into our target database without having to worry about transforming it. Let's just get the complete set of data over to our target database, and then work on populating it into the final structure. This is the role a staging area plays in the process, and this is what we're going to focus on in this chapter to get our feet wet with designing ETL in OWB. We're going to stage the data initially on the target database server in one location, where it will be available for loading.

Designing our staging area

The first step is to design what our staging area is going to look like. The staging area is the interim location for the data between the source system and the target database structure. The staging area will hold the data extracted directly from the `ACME_POS` source database, which will determine how we structure our staging table. So let's begin designing it.

Designing the staging area contents

We designed our target structure in *Chapter 3*, so we know what data we need to load. We just need to design a staging area that will contain data. Let's summarize the data elements we're going to need to pull from our source database. We'll group them by the dimensional objects in our target that we designed in *Chapter 4*, and list the data elements we'll need for each. The dimensional objects in our target are:

- Sales

  The data elements in the Sales dimensional object are:
  - Quantity
  - Sales amount
- Date

  The data element in the Date dimensional object is:
  - Date of sale
- Product

  The data elements in the Product dimensional object are:
  - SKU
  - Name
  - List price
  - Department

Download at Boykma.Com

    - Category
    - Brand

- Store

  The data elements in the Store dimensional object are:

    - Name
    - Number
    - Address1
    - Address2
    - City
    - State
    - Zip postal code
    - Country
    - Region

We know the data elements we're going to need. Now let's put together a structure in our database that we'll use to stage the data prior to actually loading it into the target. Staging areas can be in the same database as the target, or in a different database, depending on various factors such as size and space issues, and availability of databases. For our purpose, we'll create a staging area as a single table in our target database schema for simplicity and will use the Warehouse Builder's **Data Object Editor** to manually create the table.

[This is the same technique we used to create metadata for the source structures in the `ACME_POS` SQL Server database back in *Chapter 2*. We'll get to use it again as we build our staging table.]

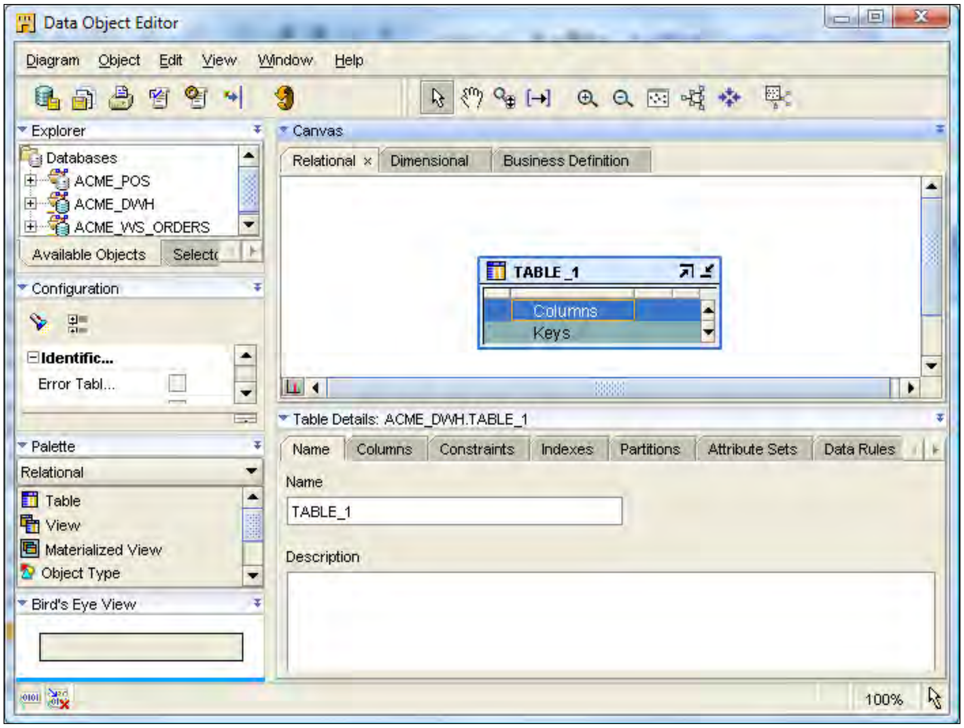
Building the staging area table with the Data Object Editor

To get started with building our staging area table, let's launch the OWB **Design Center** if it's not already running. Expand the `ACME_DW_PROJECT` node and let's take a look at where we're going to create this new table. We've stated previously that all the objects we design are created under modules in the Warehouse Builder so we need to pick a module to contain the new staging table. As we've decided to create it in the same database as our target structure, we already have a module created for this purpose. We created this module back in *Chapter 3* when we created our target user, `ACME_DWH`, with a target module of the same name.

Download at Boykma.Com

The steps to create the staging area table in our target database are:

1. Navigate to the **Databases | Oracle | ACME_DWH** module. We will create our staging table under the **Tables** node, so let's right-click on that node and select **New...** from the pop-up menu. Notice that there is no wizard available here for creating a table and so we are using the **Data Object Editor** to do it.
2. Upon selecting **New...**, we are presented with the **Data Object Editor** screen that we saw in the last chapter as well as in *Chapter 2*. However, instead of looking at an object that's been created already, we're starting with a brand-new one. It will look similar to the following:



Name: TABLE_1

Description:

3. The first tab is the **Name** tab where we'll give our new table a name. Let's call it **POS_TRANS_STAGE** for Point-of-Sale transaction staging table. We'll just enter the name into the **Name** field, replacing the default **TABLE_1** that it suggested for us.
4. Let's click on the **Columns** tab next and enter the information that describes the columns of our new table. Earlier in this chapter, we listed the key data elements that we will need for creating the columns. We didn't specify any properties of those data elements other than the name, so we'll need to figure that out.

Download at Boykma.Com

One key point to keep in mind here is that we want to make sure the sizes and types of the fields will match the fields we want to pull the data from. If this is taken care of, we won't end up with any possible overflow errors generated by the database when trying to stuff a field with a value that is too large to fit into it, or any incompatible type errors.

Eventually, we know that we're going to have to use this new table that we're building as a source when we load our final target structure. This means we'll have to make sure our data sizes and types are compatible with our final structure also, and not just our sources. When we designed our target dimensions and cube, we made sure to specify correct sizes and types and so we shouldn't face any problem here. We can't change the source columns as they are fixed, which is another important consideration. The targets right now are only defined in metadata in the Warehouse Builder, so we can easily update them if needed.

The Warehouse Builder will actually tell us if we have a problem with the data types and field lengths when we use this table in a mapping either as a source or target table. It knows the size and type of the fields in the sources and targets because we imported or created tables to represent the sources, and it does a comparison internally. It will tell us if we're trying to map something too big for a field, or to a field of an incompatible data type. We don't want to have to wait until then to specify the correct size and type, so we'll create them accordingly now.

The following will then be the column names, types, and sizes we'll use for our staging table based on what we found in the source tables in the POS transaction database:

```
SALE_QUANTITY NUMBER(0,0)
SALE_DOLLAR_AMOUNT NUMBER(10,2)
SALE_DATE DATE
PRODUCT_NAME VARCHAR2(50)
PRODUCT_SKU VARCHAR2(50)
PRODUCT_CATEGORY VARCHAR2(50)
PRODUCT_BRAND VARCHAR2(50)
PRODUCT_PRICE NUMBER(6,2)
PRODUCT_DEPARTMENT VARCHAR2(50)
STORE_NAME VARCHAR2(50)
STORE_NUMBER VARCHAR2(10)
STORE_ADDRESS1 VARCHAR2(60)
STORE_ADDRESS2 VARCHAR2(60)
STORE_CITY VARCHAR2(50)
STORE_STATE VARCHAR2(50)
STORE_ZIPPOSTALCODE VARCHAR2(50)
STORE_REGION VARCHAR2(50)
STORE_COUNTRY VARCHAR2(50)
```

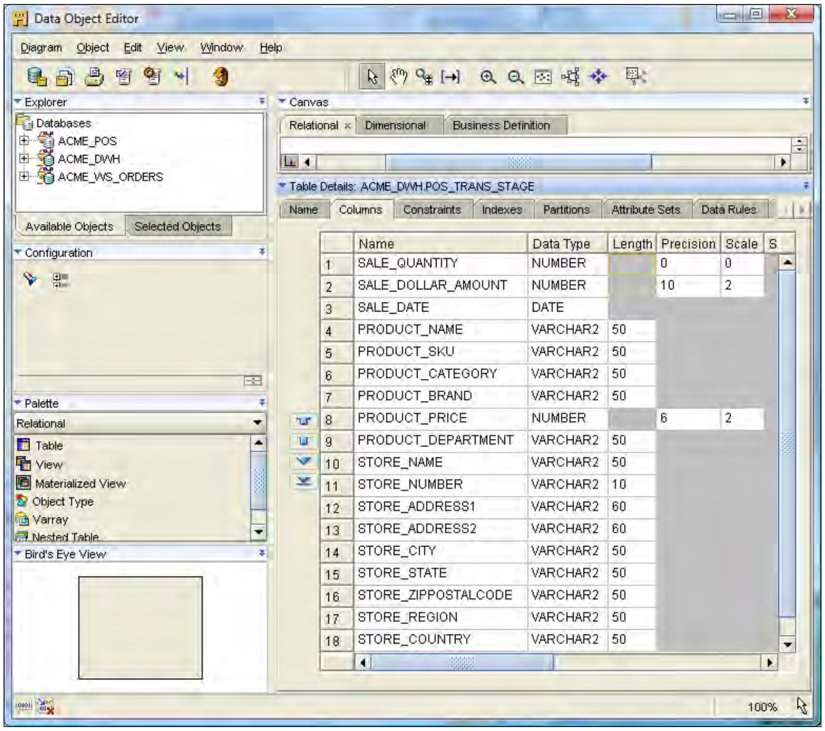
Download at Boykma.Com

There are a couple of things to note about these data elements. There are three groupings of data elements, which correspond to the three-dimensional object we created—our `Sales` cube and two dimensions, `Product` and `Store`.

> We don't have to include the dimensional object names in the data element names, but it helps to organize the data elements for eventual load into the target objects. This way, we can readily see which elements go where when the time comes to map them into the target.

The second thing to note is that these data elements aren't all going to come from a single table in the source. For instance, the Store dimension has a `STORE_REGION` and `STORE_COUNTRY` column, but this information is found in the `REGIONS` table in the `ACME_POS` source database. This means we are going to have to join this table with the `STORES` table if we want to be able to extract these two columns.

We now have the information we need to populate the **Columns** tab in the **Data Object Editor** window for our staging table. We'll enter the above column names and types into the list of columns to complete the definition of our staging table. When completed, our column list should look like the following screenshot:



	Name	Data Type	Length	Precision	Scale	S
1	SALE_QUANTITY	NUMBER		0	0	
2	SALE_DOLLAR_AMOUNT	NUMBER		10	2	
3	SALE_DATE	DATE				
4	PRODUCT_NAME	VARCHAR2	50			
5	PRODUCT_SKU	VARCHAR2	50			
6	PRODUCT_CATEGORY	VARCHAR2	50			
7	PRODUCT_BRAND	VARCHAR2	50			
8	PRODUCT_PRICE	NUMBER		6	2	
9	PRODUCT_DEPARTMENT	VARCHAR2	50			
10	STORE_NAME	VARCHAR2	50			
11	STORE_NUMBER	VARCHAR2	10			
12	STORE_ADDRESS1	VARCHAR2	60			
13	STORE_ADDRESS2	VARCHAR2	60			
14	STORE_CITY	VARCHAR2	50			
15	STORE_STATE	VARCHAR2	50			
16	STORE_ZIPPOSTALCODE	VARCHAR2	50			
17	STORE_REGION	VARCHAR2	50			
18	STORE_COUNTRY	VARCHAR2	50			

Download at Boykma.Com

5. We'll save our work using the *Ctrl+S* keys, or from the **Diagram | Save All** main menu entry in the **Data Object Editor** before continuing through the rest of the tabs. We didn't get to do this back in *Chapter 2* when we first used the **Data Object Editor**.

The other tabs in **Data Object Editor** for a table are:

- **Constraints**

  The next tab after **Columns** is **Constraints** where we can enter any one of the four different types of **constraints** on our new table. A constraint is a property that we can set to tell the database to enforce some kind of rule on the table that limits (or constrains) the values that can be stored in it. There are four types of constraints and they are:

  ◦ **Check constraint**—a constraint on a particular column that indicates the acceptable values that can be stored in the column.

  ◦ **Foreign key**—a constraint on a column that indicates a record must exist in the referenced table for the value stored in this column. We talked about foreign keys back in *Chapter 2* when we discussed the `ACME_POS` transactional source database. A foreign key is also considered a constraint because it limits the values that can be stored in the column that is designated as a foreign key column.

  ◦ **Primary key**—a constraint that indicates the column(s) that make up the unique information that identifies one and only one record in the table. It is similar to a unique key constraint in which values must be unique. The primary key differs from the unique key as other table's foreign key columns use the primary key value (or values) to reference this table. The value stored in the foreign key of a table is the value of the primary key of the referenced table for the record being referenced.

  ◦ **Unique key**—a constraint that specifies the column(s) value combination(s) cannot be duplicated by any other row in the table.

  Now that we've discussed each of these constraints, we're not going to use any for our staging table. In general, we want maximum flexibility in storage of all types of data that we pull from the source system. Setting too many constraints on the staging table can prevent data from being stored in the table if data violates a particular constraint.

Download at Boykma.Com

In this case, our staging table is a standalone table, so we don't have to worry about whether the data relates to any other tables via a foreign key. We want all the data available to our mapping, which will handle any transformations needed to make the data fit into the target system. So, no constraints are needed on this source staging table. In the next chapter, we'll have an opportunity to revisit this topic and create a primary key on a table.

- **Indexes**

  The next tab provided in the **Data Object Editor** is the **Indexes** tab. We were introduced to indexes at the end of *Chapter 4* when we discussed the details displayed for a cube in the **Data Object Editor** on the **Storage** tab. An index can greatly facilitate rapid access to a particular record. It is generally useful for permanent tables that will be repeatedly accessed in a random manner by certain known columns of data in the table. It is not desirable to go through the effort of creating an index on a staging table, which will only be accessed for a short amount of time during a data load. Also, it is not really useful to create an index on a staging table that will be accessed sequentially to pull all the data rows at one time. An index is best used in situations where data is pulled randomly from large tables, but doesn't provide any benefit in speed if you have to pull every record of the table.

Indexes are automatically created for us by the database in certain situations to support constraints. A primary key will have an index backing it up, consisting of the primary key column(s). A unique key is implemented with a unique index on the columns specified for the key. So if we were looking at creating indexes on a regular table, we would already have some if we'd specified these constraints. This is just something to keep in mind when deciding what to index in a table.

- **Partitions**

  So now that we have nixed the idea of creating indexes on our staging table, let's move on to the next tab in the **Data Object Editor** for our table, **Partitions**. **Partition** is an advanced topic that we won't be covering here but for any real-world data warehouse, we should definitely consider implementing partitions. A partition is a way of breaking down the data stored in a table into subsets that are stored separately. This can greatly speed up data access for retrieving random records, as the database will know the partition that contains the record being searched for based on the partitioning scheme used. It can directly home in on a particular partition to fetch the record by completely ignoring all the other partitions that it knows won't contain the record.

Download at Boykma.Com

There are various methods the Oracle Database offers us for partitioning the data and they are covered in depth in the Oracle documentation. Oracle has published a document devoted just to **Very Large Databases** (**VLDB**) and partitioning, which can be found at `http://download.oracle.com/docs/cd/B28359_01/server.111/b32024/toc.htm`.

Not surprisingly, we're not going to partition our staging table for the same reasons we didn't index it. So let's move on with our discussion of the **Data Object Editor** tabs for a table.

- **Attribute Sets**

  The next tab is the **Attribute Sets** tab. An **Attribute Set** is a way to group attributes of an object in an order that we can specify when we create an attribute set. It is useful for grouping subsets of an object's attributes (or columns) for a later use. For instance, with data profiling (analyzing data quality for possible correction), we can specify attribute sets as candidate lists of attributes to use for profiling. This is a more advanced feature and as we won't need it for our implementation, we will not create any attribute sets.

- **Data Rules**

  The next tab is **Data Rules**. A **data rule** can be specified in the Warehouse Builder to enforce rules for data values or relationships between tables. It is used for ensuring that only high-quality data is loaded into the warehouse. There is a seperate node—**Data Rules**—under our project in the **Design Center** that is strictly for specifying data rules. A data rule is created and stored under this node. This is a more advanced feature. We won't have time to cover it in this introductory book, so we will not have any data rules to specify here.

- **Data Viewer**

  The last tab is the **Data Viewer** tab. We discussed this in *Chapter 4* when we covered the **Data Object Editor** for viewing cubes and dimensions. This can be useful when we've actually loaded data to this staging table to view it and verify that we got the results we expected.

This completes our tour through the tabs in the **Data Object Editor** for our table. These are the tabs that will be available when creating, viewing, or editing any table object. At this point, we've gone through all the tabs and specified any characteristics or attributes of our table that we needed to specify. This table object is now ready to use for mapping. Our staging area is now complete so we can proceed to creating a mapping to load it. We can now close the **Data Object Editor** window before proceeding by selecting **Diagram | Close Window** from the **Data Object Editor** menu.

Download at Boykma.Com

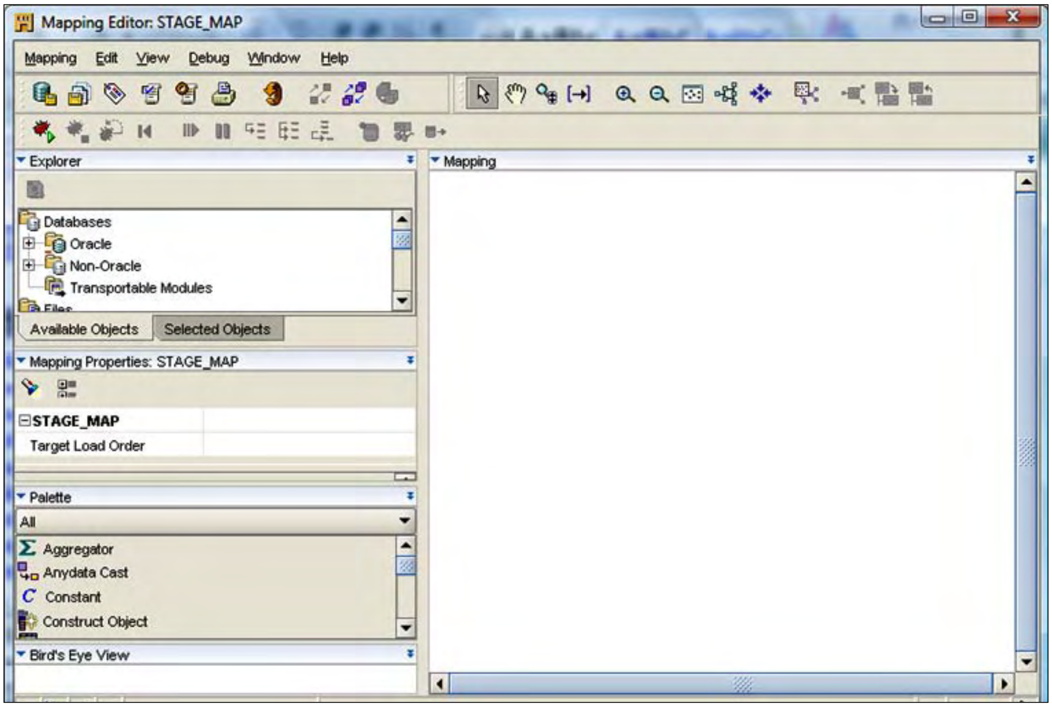
Designing our mapping

Now that we have our staging table defined, we are now ready to actually begin designing our mapping. We'll cover creating a mapping, adding/editing operators, and connecting operators together.

Review of the Mapping Editor

We were introduced to the **Mapping Editor** in the last chapter and discussed its features, so we'll just briefly review it here before using it to create a mapping. Mappings are stored in the **Design Center** under an Oracle Database module. In our case, we have created an Oracle Database module called `ACME_DWH` for our target database. So this is where we will create our mappings. In **Design Center**, navigate to the **ACME_DW_PROJECT | Databases | Oracle | ACME_DWH | Mappings** node if it is not already showing. Right-click on it and select **New...** from the resulting pop up. We will be presented with a dialog box to specify a name and, optionally, a description of our mapping. We'll name this mapping **STAGE_MAP** to reflect what it is being used for, and click on the **OK** button.

This will open the **Mapping Editor** for us to begin designing our mapping. An example of what we will see next is presented here:



Download at Boykma.Com

Unlike our first look at the **Mapping Editor** in the last chapter where we looked at the existing **DATE_DIM_MAP**, all new mappings start out with a blank slate upon which we can begin to design our mapping. We mentioned in *Chapter 4* how we'd later see other editors similar to the **Data Object Editor** when looking at mappings, and that time has arrived now.

By way of comparison with the **Data Object Editor**, the **Mapping Editor** has a blank area named **Mapping** where the **Data Object Editor** called it the **Canvas**. It basically performs the same function for viewing and laying out objects.

The **Mapping Editor** has an **Explorer** window in common with the **Data Object Editor**. This window performs the same function for viewing operators/objects that can be dragged and dropped into our mapping, and the operators we've already defined in our mapping.

Below the **Explorer** window in the **Mapping Editor** is the **Properties** window. In the **Data Object Editor** it was called **Configuration**. It is for viewing and/or editing properties of the selected element in the **Mapping** window. Right now we haven't yet defined anything in our mapping, so it's displaying our overall mapping properties.

Below the **Properties** window is the **Palette**, which displays all the operators that can be dragged and dropped onto the **Mapping** window to design our mapping. It is just like the **Palette** in the **Data Object Editor**, but differs in the type of objects it displays. The objects in this case are specific to mappings, so we'll see all the operators that are available to us.

Creating a mapping

What we just saw was a brief review of the **Mapping Editor**. Now let's begin to use it to design our mapping of the staging table. In designing any mapping in OWB, there will be a source(s) that we pull from, a target(s) that we will load data into, and several operators in between depending on how much manipulation of data we need to do between source and target. The layout will begin with sources on the left and proceed to the final targets on the right of the canvas as we design it. Right now we know that we have to pull data from the `ACME_POS` transactional database in SQL Server as our source and load it into the `POS_TRANS_STAGE` table that we just defined as our target. So let's begin by including these objects into our mapping.

Download at Boykma.Com

We need to look at the source data and determine what tables we will need to pull the data from so that we know which of the objects to include in our mapping. We first looked at the source data for the POS transactional database back in *Chapter 2.* So if you need to refresh your memory about what that looked like, now would be a good time to go back and review that quickly before moving on. In the next section, we'll start by adding a source table.

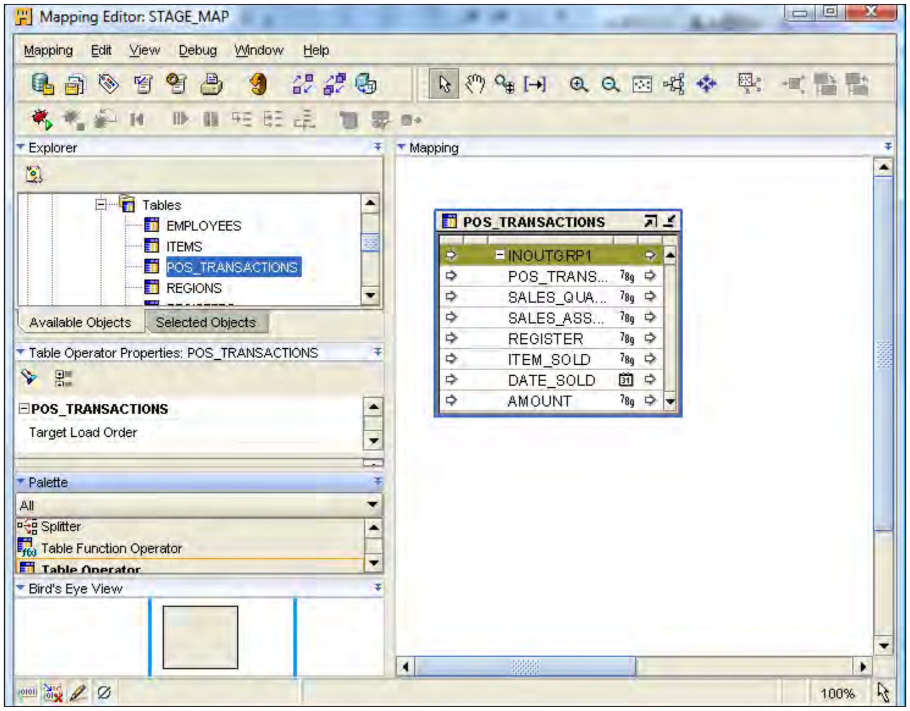
Adding source tables

We know that the first piece of information we need for loading into our staging table is the sales data—the quantity and dollar amount of each sale, and the date of the sale. Looking at our `ACME_POS` source database, we know that data is stored in the `POS_Transactions` table. Therefore, we'll start our mapping by including this table.

There are a couple of ways we can add a table to our mapping. One way is to use the **Explorer** window and the other way is to use the **Palette** window. Which one we choose is really just a matter of preference.

In the **Explorer** window, we will use the **Available Objects** tab to find the table that we want to include in our mapping. To find an object in the **Explorer**, we have to know what module it is located under. In our case, we know the **POS_TRANSACTIONS** table is defined under the **ACME_POS** module. So let's navigate to the **Databases | Non-Oracle | ODBC | ACME_POS** node in the **Available Objects** tab to find the **POS_TRANSACTIONS** table entry. Click and hold the left mouse button on **POS_TRANSACTIONS**, drag it over to the **Mapping** window, and release the left mouse button to drop the table into our mapping. Our **Mapping Editor** window should now look similar to the following:

Download at Boykma.Com



There are a couple of items to note about how the **Mapping Editor** window looks. The **Properties** window no longer shows the mapping information. It has changed to show the properties of the **POS_TRANSACTION** table as it is now highlighted in the **Mapping** canvas window.

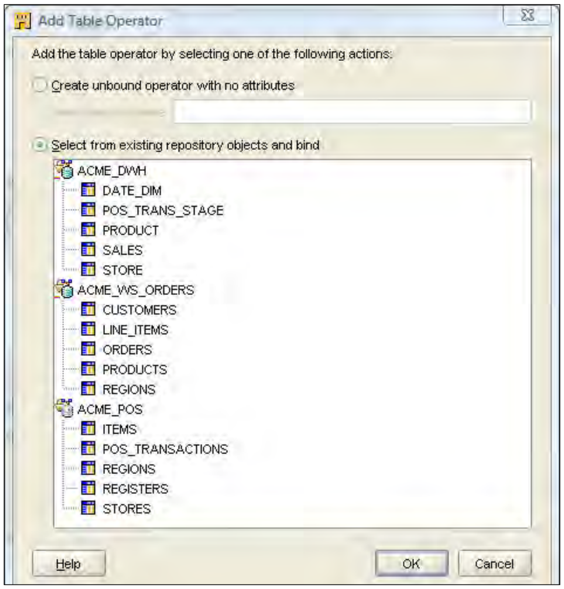
If your **Properties** window does not show the **POS_TRANSACTIONS** properties, simply click on the **POS_TRANSACTIONS** operator in the **Mapping** window. Make sure you click on the title bar of the operator window because if you click inside the window, it will select one of the attributes or groups and display the properties for that instead of displaying the properties for the operator as a whole.

Another item to note is that now we have an object in our **Mapping** window instead of a blank canvas. These objects that make up a mapping are called **operators**. In this particular case, it is a Table operator that we have placed into the mapping to represent the `POS_Transactions` table.

Download at Boykma.Com

Having just one operator in our mapping is not enough, so let's try including the remainder of the tables we'll need from our source database. We clearly need some product information to fill in. From our analysis in *Chapter 2* of the `ACME_POS` source database structures, we know that product information comes from the `Items` table. We'll include that table in our mapping now, but instead of using the **Explorer** window as we did for the **POS_TRANSACTIONS** table, let's see how the **Palette** window works for including objects into our mapping.

The operators in the **Palette** are sorted alphabetically, so we'll scroll the window until we see the **Table Operator**. Click and drag the **Table Operator** from the **Palette** window onto the **Mapping** window. As soon as we drag it onto the **Mapping** window, we are presented with a pop up like the following screenshot:


Add Table Operator

Add the table operator by selecting one of the following actions:

Create unbound operator with no attributes

Select from existing repository objects and bind

- ACME_DWH
  - DATE_DIM
  - POS_TRANS_STAGE
  - PRODUCT
  - SALES
  - STORE
- ACME_WS_ORDERS
  - CUSTOMERS
  - LINE_ITEMS
  - ORDERS
  - PRODUCTS
  - REGIONS
- ACME_POS
  - ITEMS
  - POS_TRANSACTIONS
  - REGIONS
  - REGISTERS
  - STORES

This pop up asks us which table we want to include as this table operator. We have a couple of options offered to us. We can create an **unbound operator**, which has no attributes, in other words, it is a blank table that we can define as we like, or we can specify an existing table from our project. An unbound operator is one which is not associated with (that is, bound to) an existing database object. The act of binding in OWB associates a generic operator with an actual defined object in the project. When we dragged our **POS_TRANSACTIONS** table from the **Explorer** window, it did not ask us about this because we started with a specific named table. The operators in the **Palette** window are all generic and are not associated with any specific object of that type. With unbound operators, you can actually use the **Mapping Editor** to create a data object. We'll actually get to do this in the next chapter when we have to create a lookup table.

Download at Boykma.Com

We're going to stick to the objects already defined for our operators. So we're going to select the **ITEMS** table under the **ACME_POS** entry in the list of table names that the pop-up window presents to us. We will click on the **OK** button to include the **ITEMS** table operator in our mapping. Notice how the **Add Table Operator** dialog box presents the information to us. It lists every possible table in our project and organizes them by module. `ACME_DWH` is our main data warehouse module that we created for our target in order to build our data warehouse. We can see the tables that were created for our cube and dimensions along with the staging table we created. `ACME_WS_ORDERS` is the web site order's database that we imported for source data from the web site, and `ACME_POS` is what we're working with right now to build a staging area.

Let's talk about organizing our tables in the **Mapping** window before we go any further. In general, it's always a good idea to place source operators on the left and the target operators on the right. So let's just make sure we keep the tables we're dropping into our **Mapping** window towards the left side of the window, one above the other. Click and hold on the header of the operator to drag it around.

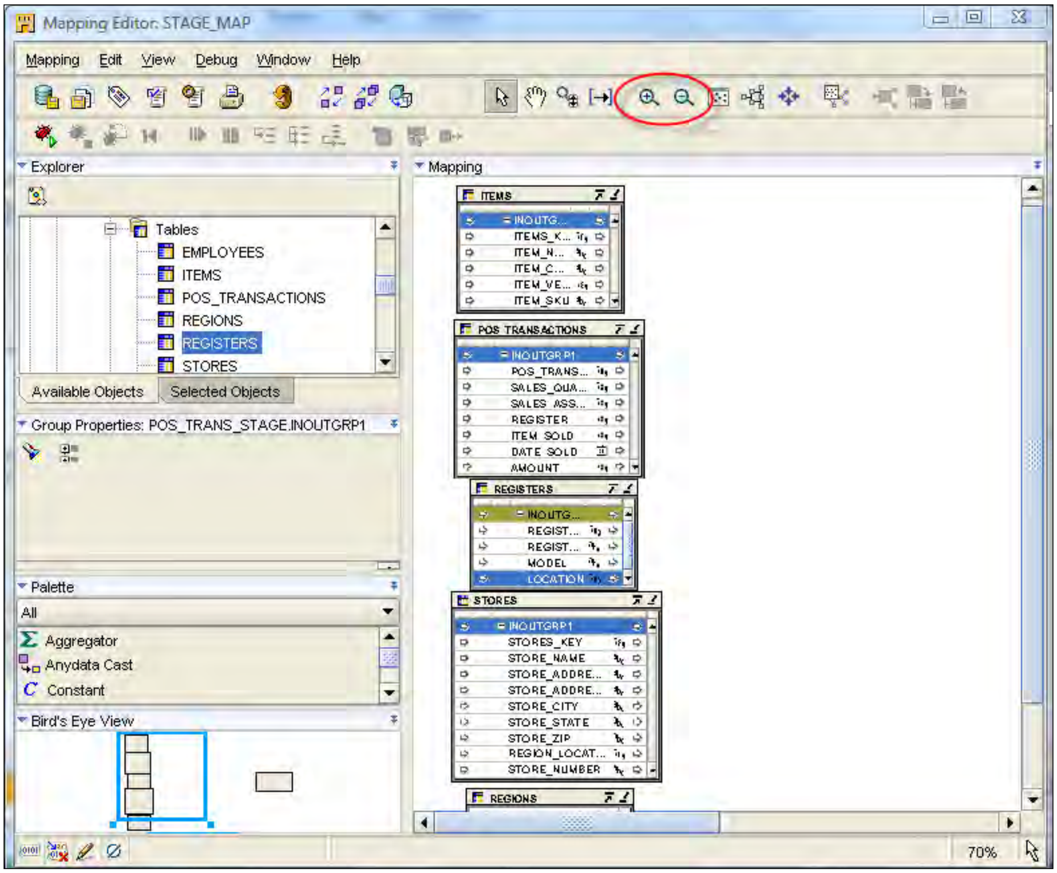
We've seen how to include a table operator in our mapping using either of the two methods. Using either one now, we'll include the remainder of the source tables that we're going to need into our mapping—the `REGISTERS`, `STORES`, and `REGIONS` tables. It should be clear why we need the `STORES` and the `REGIONS` tables. These tables contain information about each store that we'll need, including the address, region, and country. But why do we need the `REGISTERS` table? We're not going store any information about the register used. From our analysis back in *Chapter 2*, we saw that the register information pointed to the store where the register was located in and the main `POS_TRANSACTIONS` table only had a foreign key column for the register—not the store or region. This information was kept in separate tables with a foreign key to the store, which is stored in the `REGISTERS` table. So, if we hope to be able to retrieve the store and region information out of the source database, we're going to need the `REGISTERS` table to get there.

Now go ahead and include those three tables into the mapping using either of the two methods we just discussed. When that is completed, make sure the tables are organized vertically on the left side of the mapping window in following order: the `ITEMS` operator on top, then `POS_TRANSACTIONS`, then the `REGISTERS` operator next, then the `STORES` operator, and then the `REGIONS` operator at the bottom. We'll see soon why we are paying attention to the order in which we display the operators in the **Mapping** canvas.

Download at Boykma.Com

You'll notice that while dragging objects around the **Mapping** window, it will grow in size automatically to hold the objects we are placing there. So we needn't be too concerned if our source tables are not all the way over to the left. When we place our target table, we'll put it to the righthand side of the sources. Just make sure that the source tables are all together.

Your **Mapping Editor** window should now look similar to the following:



The five source tables may not all be visible at once as we saw in the previous screenshot. The **Mapping** view has been zoomed out, so mostly all are visible here. However, there is a trade-off in readability the more we zoom out. You can manipulate the zoom yourself to find a size that's comfortable for your viewing. The zoom buttons in the toolbar have been circled in the screenshot we just saw. Click on the magnifying glass with the plus sign to zoom in and the minus sign to zoom out.

Download at Boykma.Com

Now that we have our source table all included and laid out in the **Mapping** window, we'll move on to discuss getting our target included in the mapping.

Adding a target table

Let's now turn our attention to the target for this particular mapping. As this is a staging-related mapping, we're going to be loading our staging table and so that will become our target. Let's find the **POS_TRANS_STAGE** table in the **Explorer** window. We'll navigate to **Databases | Oracle | ACME_DWH | Tables | POS_TRANS_STAGE** in the **Explorer**, and click and drag the **POS_TRANS_STAGE** table to the righthand side of our source tables in the **Mapping** window. Let's leave some space between the source table and the target tables. We'll shortly see the reason behind this when we start connecting our source to the target.

Connecting source to target

The process of connecting the source to the target is the means of telling the Warehouse Builder which data fields from the source go in which data fields in the target. We might be tempted to just connect the data fields from the source tables directly to the corresponding fields in the target. For instance, we know that the `ITEMS` table has the `ITEM_NAME` field, which needs to be stored in the target in the `PRODUCT_NAME` column; so why can't we just connect the two directly?

The reason we can't connect the two directly is because we have to keep in mind what that means in terms of mapping. If we just connect a line from a source table attribute to a target table attribute, we're telling the Warehouse Builder there is a one-to-one mapping from source to target. This means we can read a record from the source table and store the column values directly in the target table with no need of further manipulation. The problem in our case is that we're including information in our target table from multiple source tables, and the Warehouse Builder is not going to let us connect multiple source tables to a single target table directly as it won't know how to combine the data.

In addition to the joining of the tables, connecting directly from source to target would also imply that the data was at the granularity we need. We discussed granularity (or the level at which the data is stored) in *Chapter 3* and decided that our warehouse would store the data by product, store, and date. However, looking at our source data, we can see that the data is actually stored by register. This is actually a lower level of detail than we need, so we'll need to sum up the data for each register in a store to get the total for the store.

Download at Boykma.Com

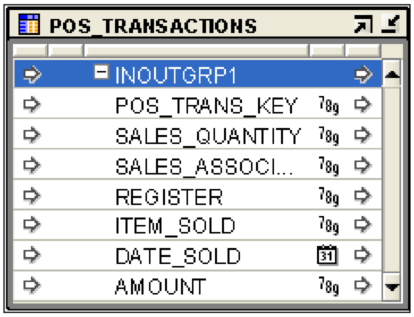
This means we will have to provide some kind of intervening operators to get the data combined from the five source tables into the one target table, summed by product, store, and date. But what operators are we going to use? Remember our discussion in *Chapter 5* that introduced us to the various operators available in the Warehouse Builder. We particularly mentioned in the *Data flow operators* section that directly connecting source to target would only work for a one-to-one mapping between a source table and a target table. Then went on to discuss some of the operators that can be used for data flows, and one of them was a **Joiner** operator. If you re-read the explanation, you'll see that a joiner is exactly what we need in this case because we have to take multiple source tables and combine (or join) them into one record in the target. We also discussed an **Aggregator** operator that can be used to aggregate data. In this case, we need to sum data at a higher level before storing it. So this should be exactly the operator we need for that purpose.

Now that we've settled on the two data flow operators we need, let's place them into our mapping between the sources and the target. Now you can see why we left some space between the sources and the target. Scroll down through the **Palette** window until the **Joiner** operator is visible, drag this operator into the **Mapping** window, and drop it between the sources and target.

Joiner operator attribute groups

We were introduced to the concept of attribute groups in the last chapter when we were looking at `DATE_DIM_MAP` in the **Mapping Editor**. It's time to talk about attribute groups again, because we can see that the **Joiner** operator has three groups defined, but the attributes in our table operators are all in one group. The groups in operators we saw are generally input groups, output groups, or both.

In our table operators we can see that there is only one group as we mentioned, called **INOUTGRP1**. The following screenshot is an example of using the **POS_TRANSACTIONS** table operator:



Download at Boykma.Com

[There are actually two clues to identify the attributes that can be used for both input and output: one is the name of the group, which has both `IN` and `OUT` in it, and the second is the little arrows that appear on each attribute line—one arrow on the left pointing in for input and one on the right pointing out for output.]

If an attribute is in an input group, then we can connect an attribute from another operator on the left to let the data flow from that attribute to this one. These connections always enter an operator from the left. Now we can see why it's a good idea to put our sources on the left and targets on the right. This is the direction the data flows through the operators.

If the attribute can be used for output in either an output group or an in/out group, then it means the data from the attribute can be used as input into another operator. Output from an operator always flows out from the right side of the operator.

If we look at the **Joiner** operator we just dropped into our mapping, we can see that there are no attributes defined in any of the three groups. Before we add attributes, let's talk briefly about the groups in a **Joiner** operator. By default, the operator is created with two input groups and one output group. Each input group corresponds to a separate table or other data operator, and the output group represents the combined (joined) output from the input tables.

We have five source tables to join together, but this Joiner operator has only two input groups. Have no fear; a Joiner can have more than two input groups. We have to edit this Joiner to add three more input groups. To edit it, right-click on the header of the box and select **Open Details...** to open the **Joiner Editor**. This dialog box will allow us to edit the number of groups as well as change the group names if we want something different from **INGRP1** and **INGRP2**.

The **Joiner Editor** can be used to edit not only the groups, but also the attributes that compose each group. So if we right-click inside the **Joiner** box on a group and select **Open Details...**, we will get the same dialog box with just the individual tab selected that corresponds to the group we clicked on.

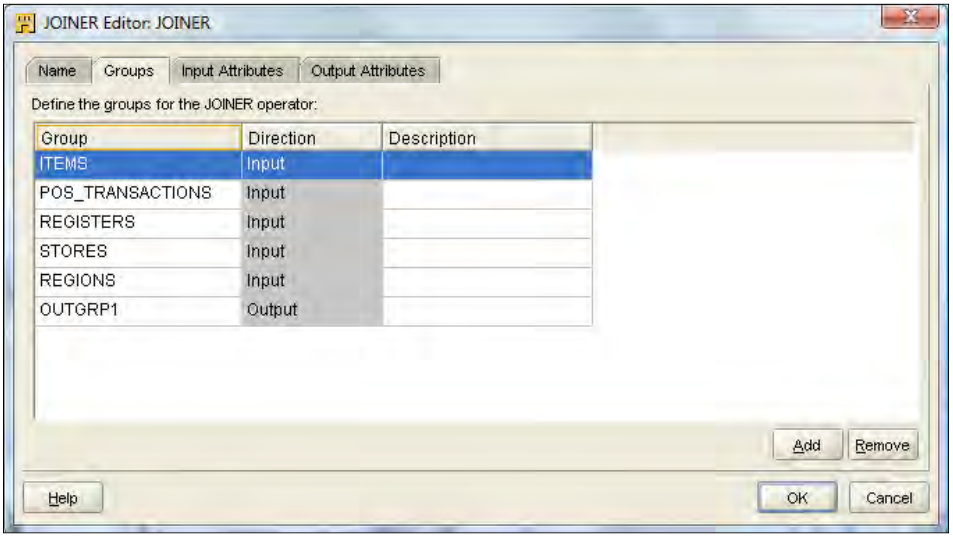
With the **Joiner Editor** open, let's click on the **Groups** tab. We'll click three times on the **Add** button in the lower-right corner to add three more groups. Notice that the default names it assigns are `INGRP3`, `INGRP4`, and `INGRP5`. This means it knew we wanted input groups and not output groups. Did it read our minds? Well, not exactly. As it turns out, Joiners can have only one output group and no combined input/output groups. So the only kind of group left to add when we click on the **Add** button is an input group.

Download at Boykma.Com

While we're on the **Groups** tab, let's modify the names of our input groups. For a Joiner where each input group corresponds to a table, it makes logical sense to rename the input groups to reflect the name of the table that is being input by that group. We'll leave the output group with the default name. By just clicking on the name of the input group, we can type in a new name. So, let's rename each input group as follows:

- `INGRP1` to `ITEMS`
- `INGRP2` to `POS_TRANSACTIONS`
- `INGRP3` to `REGISTERS`
- `INGRP4` to `STORES`
- `INGRP5` to `REGIONS`

After renaming this is how it will look:



Define the groups for the JOINER operator:

Group	Direction	Description
ITEMS	Input	
POS_TRANSACTIONS	Input	
REGISTERS	Input	
STORES	Input	
REGIONS	Input	
OUTGRP1	Output	

Now we'll click on the **OK** button to close out the **Joiner Editor** dialog box.

Download at Boykma.Com

You may be wondering if there is a reason why we picked a particular sequence of tables and paid attention to the order in which we displayed the table operators on the canvas. It's to match the sequence of tables that we set up in the **Mapping** window from top to bottom as displayed in the previous image. However, this is not a hard and fast requirement; they could be in any order. To ensure a good appearance, that is, keep lines from crisscrossing all over the mapping when we're done with it, it's a good idea to plan ahead and use the same order. We'll see this later.

Also, we don't have to worry about capitalization as the editor will automatically put everything we type into uppercase.

Connecting operators to the Joiner

Now that we have our Joiner groups defined, it's time to start making some connections between operators. The act of connecting operators is a matter of clicking and dragging a line from an output attribute of one operator to an input attribute of another operator, or from one output group in one operator to an input group in another operator. If we connect two attribute groups together, we're telling the **Mapping Editor** to go ahead and connect every attribute in the group. If we have several attributes, this is a convenient way to connect them. So, click and drag **INOUTGRP1** of the **ITEMS** table operator onto the **ITEMS** group of the **JOINER**. Immediately, it will add all the attributes from the **ITEMS** table to the **ITEMS** group in the **JOINER** and connect each one with a line.

Alternatively, we could have clicked and dragged a line from each attribute in the **ITEMS** table and dropped it on the **ITEMS** group in the **JOINER**. But it was quicker and more efficient to drag the entire group even though we won't be using every attribute.

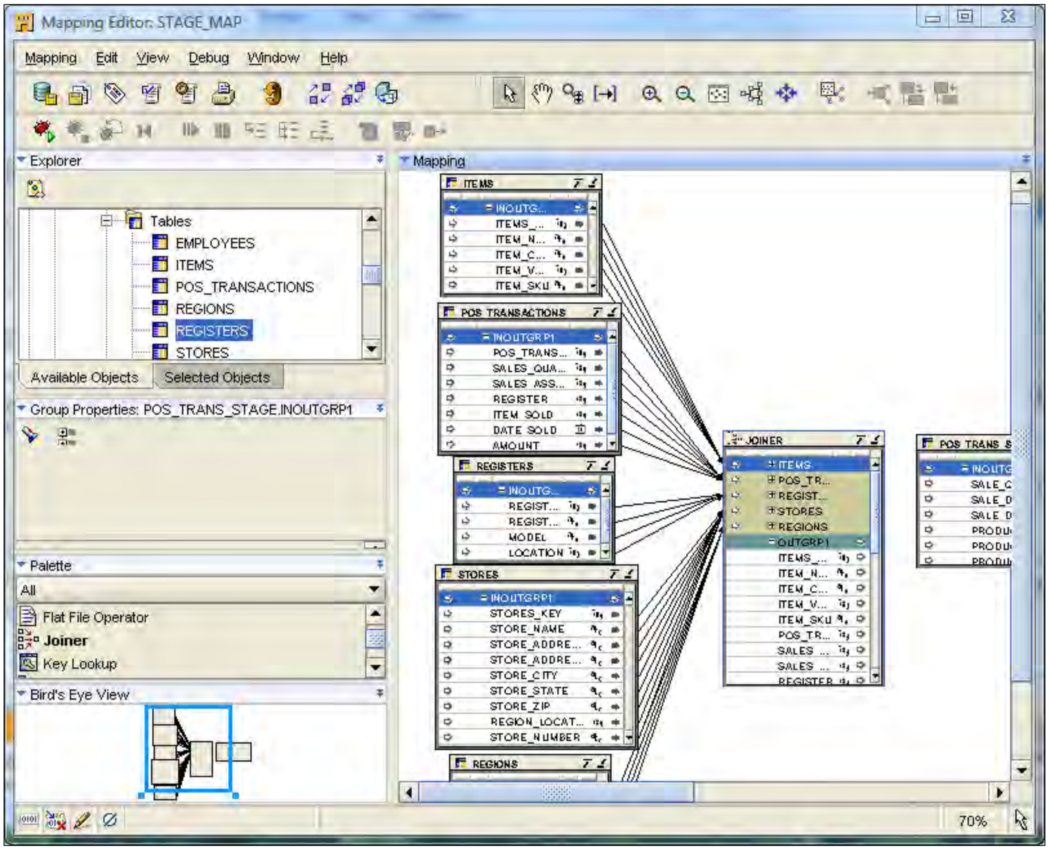
Leaving attributes in the **JOINER** that we're not going to use will not affect the final result of our mapping. We're just going to drag the attributes we're going to need over to the target table in a moment. OWB will ignore the attributes we don't use when it builds its underlying code.

Notice that if we now scroll down our **JOINER** operator to where the **OUTGRP1** is visible, we can see that it automatically added attributes to the output group corresponding to each of the input group attributes.

Download at Boykma.Com

To better view the attributes in the **JOINER** operator, we can make the box bigger by clicking and dragging the border of the box to a bigger size, either vertically or horizontally. Also, if we don't want to see all the attributes, we can click on the minus sign on a group to collapse it so that the attributes are not visible. Clicking on the plus sign then restores the attributes.

Now let's repeat the same procedure with the **POS_TRANSACTIONS**, **REGISTERS**, **STORES**, and **REGIONS** tables. That is, let's drag the **INOUTGRP1** group to the corresponding group in the **JOINER** for each table to connect them. Feel free to click on the minus sign on each group to collapse it to get a better view of the next group. If the ordering of the tables has been maintained between the **JOINER** groups and the table operators, and all the input groups of the **JOINER** have been collapsed (which is why the attributes are not visible), the mapping should look similar to the following screenshot:



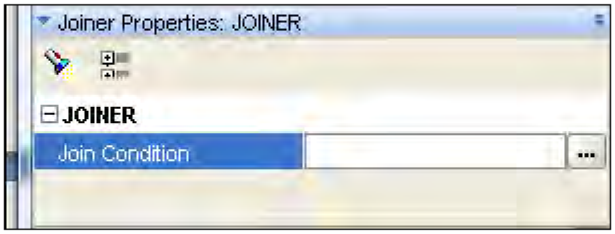
Download at Boykma.Com

Defining operator properties for the Joiner

The next step in the process is to specify how we want the tables joined. We need to identify the attributes to use in the join condition. We will do that by modifying a property of the **JOINER**. This will be our first experience of working with the **Properties** window in the **Mapping Editor**. If the **JOINER** operator is not already selected, click once on the header of the box to select it and the **Properties** window will immediately change to display the properties of the selected object; in this case it's **JOINER**. We can see one property mentioned there, **Join Condition**.

[If you click inside the **JOINER** operator on a group or attribute, the **Properties** window will display the properties for that group or attribute and not the **JOINER** operator as a whole. We want the **JOINER** properties, so make sure the **JOINER** itself is highlighted by clicking on the header of the **JOINER** window.]

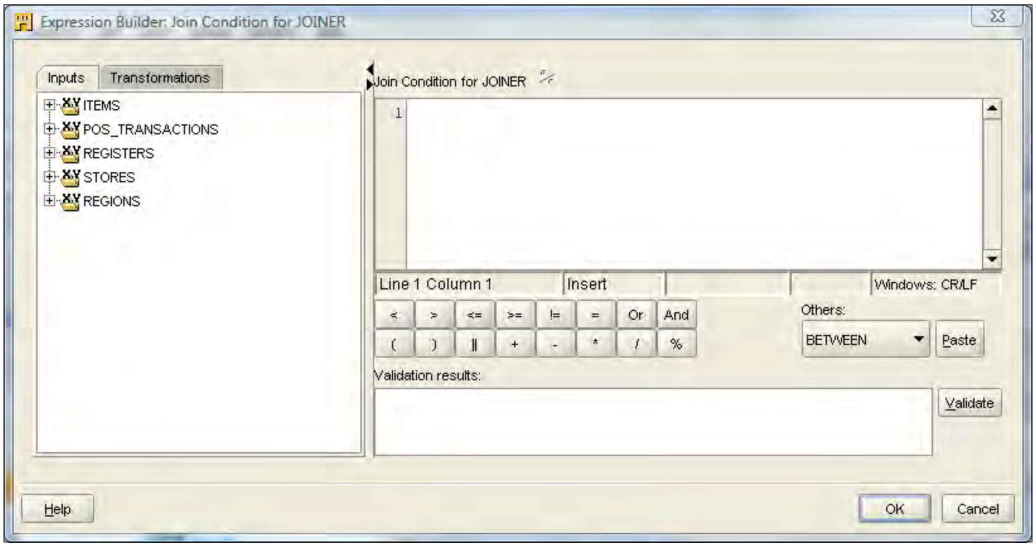
Click on the blank box to the right of the **Join Condition** label. The **Properties** window will now look like the following screenshot, which is ready for our input of the join condition:



Now we can type the join condition directly in the white box. This can get a bit tedious, especially as we'd need to know the correct syntax for specifying attributes. The Warehouse Builder refers to attributes by group as well as attribute name. So, to help us out with this, OWB provides the **Expression Builder**. This is a dialog box we can invoke to interactively build our Joiner condition.

Download at Boykma.Com

We can invoke **Expression Builder** by clicking on the button with the three dots (**...**) to the right of the blank white box. It looks like the following screenshot before anything is filled in:

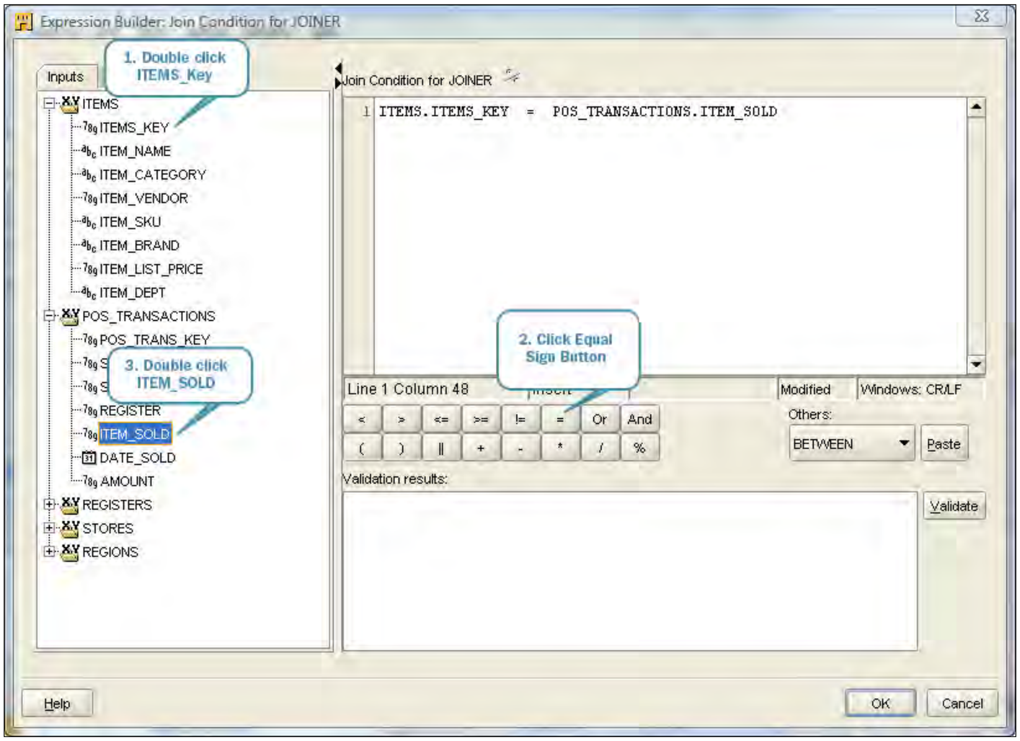


Notice on the left the list of input groups in the **JOINER**. From this list we're going to select the attributes we need and will include them in our expression in the correct format. We just need to make sure we specify the correct attributes and the correct join relation, which will be the equal to (=) symbol in our case.

We do need to know a little about SQL join syntax at this point. The **Expression Builder** provides us with the list of attributes and the relational operator buttons, which will insert the indicated relations. However, we need to insert them in the right order. Fortunately, the syntax is not very complicated. We just need to specify which column from one table equals which column from the table being joined to. Then include an equality for each table with each of the equalities separated by `AND`.

Download at Boykma.Com

We've seen the ITEMS table from our analysis in *Chapter 2* of the source data structures in the ACME_POS database. So we know that the POS_TRANSACTIONS table contains a foreign key field pointing to the record in the ITEMS table for the particular item for that transaction. This gives us clues about which columns will be needed in the first join equality—the ITEM_SOLD attribute from POS_TRANSACTIONS and the ITEMS_KEY attribute from the ITEMS operator. So, we'll expand the **ITEMS** group on the left and double-click the **ITEMS_KEY** attribute to add it to the expression. As we want every record included where the **ITEM_SOLD** equals the **ITEMS_KEY**, we will include an equal sign next by clicking on the button with the **=** sign on it. We'll finish this first relation by expanding the **POS_TRANSACTIONS** group and double-clicking on the **ITEM_SOLD** attribute to include it. Our expression now looks like the following with the steps highlighted with callouts:



```
ITEMS.ITEMS_KEY  =  POS_TRANSACTIONS.ITEM_SOLD
```

The attributes include the group name first. This is the syntax it uses to identify the specific attribute. It's common for the same attribute name to appear in more than one group, and this syntax will make it explicit which attribute is being referred to.

Download at Boykma.Com

We're not done yet, because so far we've only accounted for two of the four tables in our join condition. We also need to include `REGISTERS`, `STORES`, and `REGIONS` tables. The `REGISTER` attribute of the `POS_TRANSACTIONS` group contains the foreign key to the `REGISTERS_KEY` attribute of the `REGISTERS` table, so let's add that one. But before we add it, we need an `AND`. So let's click on the **And** button (which is near the button labeled with an equal to sign) to enter it into our mapping, and then press the *Enter* key to advance to the next line. We move to a new line to prevent our expression from extending past the viewable window. The expression could extend past it and still work. But for ease of viewing, we'll enter it vertically instead of horizontally so that we don't have to scroll.

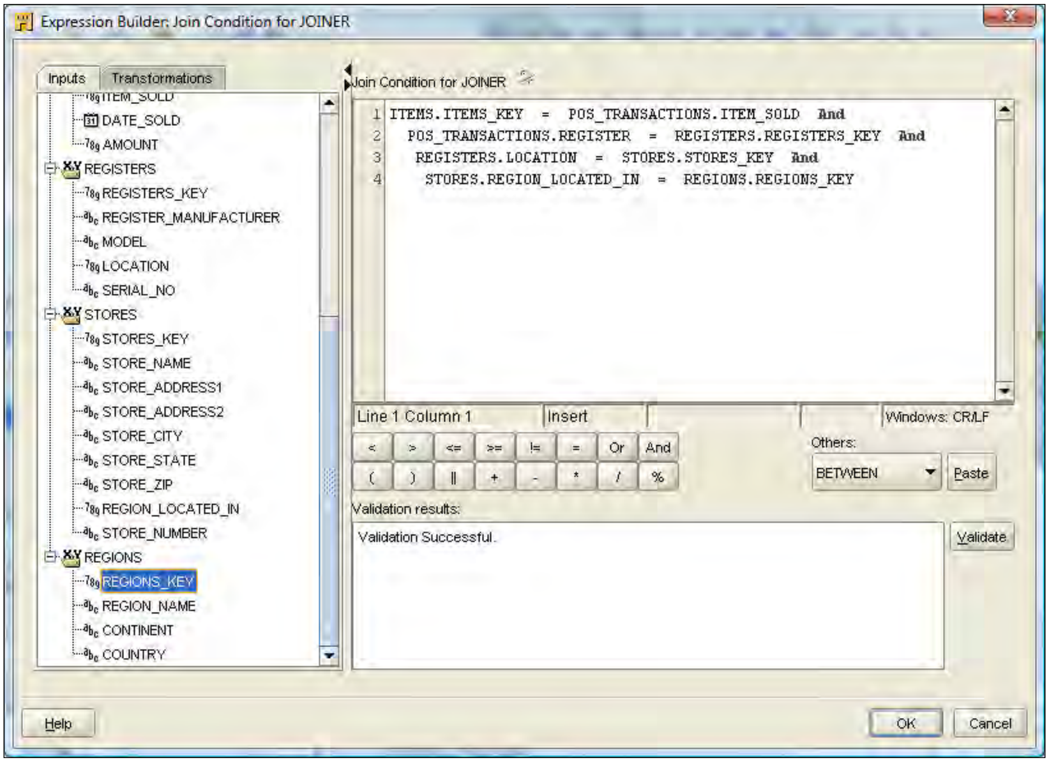
Now we'll enter:

1. The **REGISTER** attribute by double-clicking on it in the **POS_TRANSACTIONS** group.
2. The equal to sign by clicking on the corresponding button.
3. The **REGISTERS_KEY** attribute by double-clicking on it under the **REGISTERS** group.
4. This expression is followed by another `AND` by clicking on the **And** button.
5. Press the *Enter* key.

The equal to sign and the `And` can be typed in manually if you prefer, rather than having to click on the buttons each time. Some people find that quicker to enter, and will just double-click the attributes needed and manually type the other operators that are needed.

Continue in the like manner with the `LOCATION` in the `REGISTERS` group equal to `STORES_KEY` in the `STORES` group, and `REGION_LOCATED_IN` in the `STORES` group equal to the `REGIONS_KEY` in the `REGIONS` group. Do not include **And** after this last part of the expression as it is the end. When completed, the expression should look like the one in the following screenshot:

Download at Boykma.Com



```
ITEMS.ITEMS_KEY  =  POS_TRANSACTIONS.ITEM_SOLD  And
  POS_TRANSACTIONS.REGISTER  =  REGISTERS.REGISTERS_KEY  And
   REGISTERS.LOCATION  =  STORES.STORES_KEY  And
    STORES.REGION_LOCATED_IN  =  REGIONS.REGIONS_KEY
```

Validation Successful.

We can click on the **Validate** button now to make sure the expression we just entered is a valid expression, meaning that we used the correct SQL syntax. We should get the **Validation Successful** message.

Depending on what release number of the database you are running, you may get the following error message instead of the success message: **An error occured during expression validation. Bad expression return type**. This is a known bug, ID 7417869, which is reported to have occurred in Release 10.2.0.4 of the database and is fixed in 10.2.0.5. It also reports that the mapping will deploy and execute successfully even if this validation bug occurs. The bug report also says the error occurred in Release 11.1.0.7 of the database, which is the most recently released version available to licensed users of the database. However, it works in v11.1.0.6, which is the most recent publicly available version for download that we are using for this book. The screenshot we just saw was taken from 11.1.0.6 and we can see the **Validation Successful** message. So if you are using this version, you should not see the error.

Download at Boykma.Com

Click on the **OK** button and we are done specifying the join condition for the **JOINER** operator. We can now see that it filled in the join condition text into the **Join Condition** entry in the **Properties** window. This completes the Joiner, but we still have to aggregate the data so that it's at the level we need for loading into the data warehouse. For aggregating data, we'll now include an **Aggregator** operator.

Adding an Aggregator operator

An Aggregator operator is used to apply an aggregate function to the data. Aggregation functions are documented in the *Oracle Database SQL Language Reference* at `http://download.oracle.com/docs/cd/B28359_01/server.111/b28286/functions001.htm#i89203`. *Chapter 5* of this document has a section devoted to aggregate functions. In our case, we will need to add up the sales quantities and dollar amounts for every product, and the store and date combination to have data at the right level to load into the data warehouse. For this aggregation of data, we can use a `SUM()` function.

The Aggregator operator requires that we specify a few things for it to function correctly. As with any operator, there are attribute groups to set and an Aggregator operator has one input and one output group. For the input group, we will drag the output attributes of the Joiner operator. We have to specify a **group by clause** that the Aggregator operator is going to use to group the data, and it will create an output attribute for every attribute we use in the group by clause. We have to manually add output attributes for any of the values that are going to be summed up, and then specify the `SUM()` function to use for them.

We'll follow a similar process to add the Aggregator operator as we did for the Joiner; drag an **AGGREGATOR** operator onto the canvas to the right of the **JOINER** operator, connect output attributes from the **JOINER** operator to the input of the **AGGREGATOR** operator, define properties for the **AGGREGATOR** operator, and then connect the output of the **AGGREGATOR** operator to the **POS_TRANS_STAGE** table operator. Here we'll outline the steps to follow without going into as much detail as we did for the Joiner operator:

1. Drag an **AGGREGATOR** operator from the **Palette** window to the canvas and drop it to the right of the **JOINER** operator between that operator and the **POS_TRANS_STAGE** target operator. You may have to move the **POS_TRANS_STAGE** target operator further to the right to make enough room.
2. Connect the output attributes from the **JOINER** operator as input to the **AGGREGATOR** operator by dragging the **OUTGRP1** output group and dropping it on the **INGRP1** input group of the **AGGREGATOR** operator. This will map every output attribute at once, so we don't have to do each one individually.

Download at Boykma.Com

We have one issue we need to address with the input attributes for the Aggregator and that is related to the `DATE_SOLD` attribute from the Joiner operator. An attribute of the `DATE` type includes both date and time of day. We are going to sum up the data by date. But if we include the time of day, we'll get multiple dates occurring on the same day that are treated as distinct because the time is different. We want the sales for every product in a store for a single date to sum together regardless of the time of day the sale occurred. We need to strip out the time from the `DATE_SOLD` attribute, so we just have the date as input into the Aggregator operator. For that task, we need a Transformation Operator to apply the `TRUNC()` function to the value first. We'll discuss Transformation Operators in greater depth in the next chapter, but let's use one now to take care of the date.

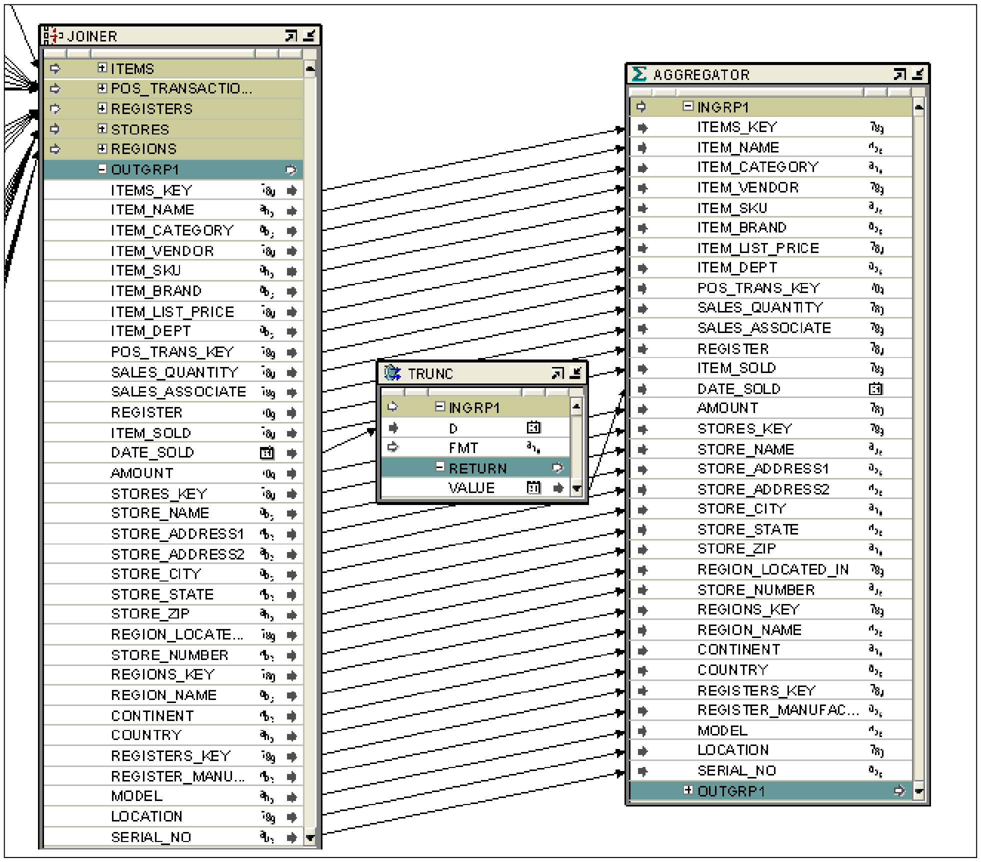
3. We need to remove the line that got dragged to the input of the **Aggregator** operator for the `DATE_SOLD` attribute by clicking on the line and pressing the *Delete* key, or right-clicking and selecting **Delete** from the pop-up menu. Make sure the correct line is selected. Attribute groups can be expanded to spread the lines apart better so that it's easier to click.
4. Drag a **Transformation Operator** from the **Palette** window and drop it on the canvas between the **Joiner** operator and the **Aggregator** operator near the **DATE_SOLD** attribute. In the resulting pop up that appears, we'll scroll down the window until the `Date()` functions appear and then select the `TRUNC()` function. It will look like the following:

   ```
   TRUNC(IN DATE, IN VARCHAR2) return DATE
   ```

   Click on that line and then click on the **OK** button to select it. It will drop a **TRUNC** Transformation Operator on the canvas.
5. Connect the **DATE_SOLD** attribute in the **OUTGRP1** group of the **Joiner** to the **D** attribute of the **INGRP1** of the **TRUNC** transformation operator. Then connect the **VALUE** attribute of the **RETURN** output group of the **TRUNC** operator to the **DATE_SOLD** attribute of the **INGRP1** group of the **Aggregator** operator.

Download at Boykma.Com

The canvas should now look similar to the following screenshot:



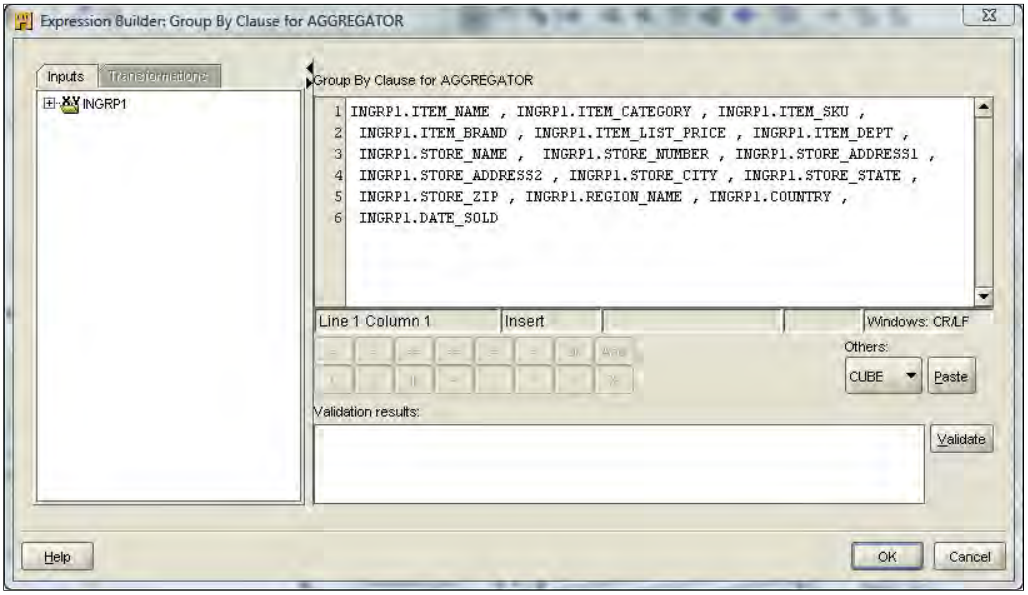
6. We have our input set for the Aggregator operator and now we need to address the output. Let's select the Aggregator operator by clicking on the title bar of the window where it says **AGGREGATOR**. The **Properties** window of the **Mapping Editor** will display the properties for the aggregator. If it doesn't, then make sure the title bar of the window was selected for the operator and not somewhere inside the operator.
7. The very first attribute listed is **Group By Clause**. We'll click on the ellipsis (...) on its right to open the **Expression Builder** for the **Group By Clause**. This is similar to how we launched it earlier to edit the join condition for the Joiner operator.

Download at Boykma.Com

8. Enter the following attributes separated by commas by double-clicking each in the **INGRP1** entry in the left window:

   ```
   INGRP1.ITEM_NAME , INGRP1.ITEM_CATEGORY , INGRP1.ITEM_SKU ,
   INGRP1.ITEM_BRAND , INGRP1.ITEM_LIST_PRICE , INGRP1.ITEM_DEPT
   , INGRP1.STORE_NAME ,  INGRP1.STORE_NUMBER , INGRP1.STORE_
   ADDRESS1 , INGRP1.STORE_ADDRESS2 , INGRP1.STORE_CITY , INGRP1.
   STORE_STATE ,  INGRP1.STORE_ZIP, INGRP1.REGION_NAME , INGRP1.
   COUNTRY , INGRP1.DATE_SOLD
   ```

   When completed, it should look similar to the following screenshot:

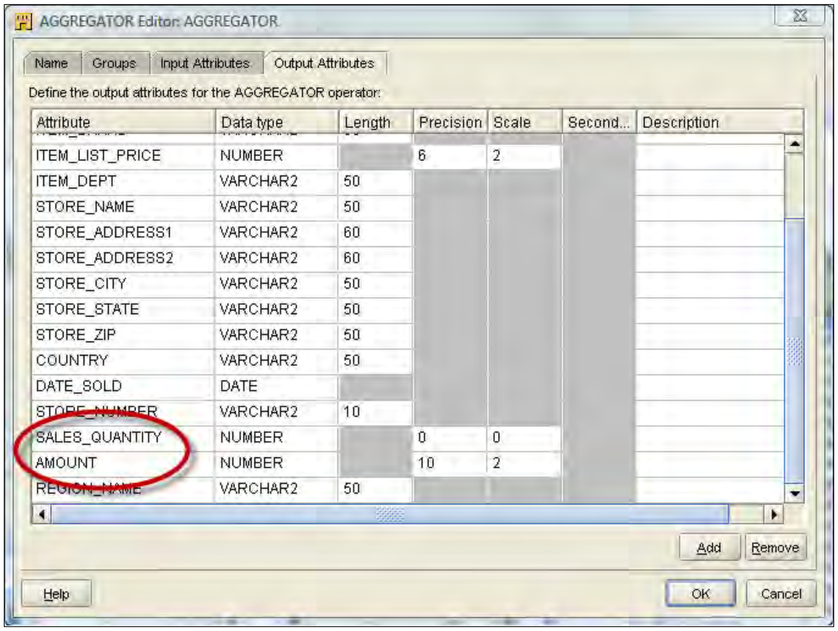


```
INGRP1.ITEM_NAME , INGRP1.ITEM_CATEGORY , INGRP1.ITEM_SKU ,
 INGRP1.ITEM_BRAND , INGRP1.ITEM_LIST_PRICE , INGRP1.ITEM_DEPT ,
 INGRP1.STORE_NAME ,  INGRP1.STORE_NUMBER , INGRP1.STORE_ADDRESS1 ,
 INGRP1.STORE_ADDRESS2 , INGRP1.STORE_CITY , INGRP1.STORE_STATE ,
 INGRP1.STORE_ZIP , INGRP1.REGION_NAME , INGRP1.COUNTRY ,
 INGRP1.DATE_SOLD
```

9. We'll click on the **OK** button to close the **Expression Builder** dialog box and looking at the **AGGREGATOR** now, we can see that it added an output attribute for each of these attributes in our group by clause. This list of attributes has every attribute needed for the **POS_TRANS_STAGE** operator except for the two number measures, `SALE_QUANTITY` and `SALE_DOLLAR_AMOUNT`. So let's add them manually.
10. We'll right-click on the **OUTGRP1** attribute group of the **AGGREGATOR** operator and select **Open Details...** from the pop up. We used this editor earlier for the Joiner to edit the groups, and now we're going to use it for the Aggregator to edit the attributes in a group.

Download at Boykma.Com

10. We'll click on the **Output Attributes** tab, and then on the **Add** button twice to add two new output attributes. Let's click on the first one it added (**OUTPUT1**) and change its name to **SALES_QUANTITY** and leave the type **NUMBER** with **0** for precision and scale. We'll click on the second one (**OUTPUT2**) and change the name to **AMOUNT** and make it type **NUMBER** with precision **10** and scale **2**. After making these changes, this is how it will look:



Define the output attributes for the AGGREGATOR operator:

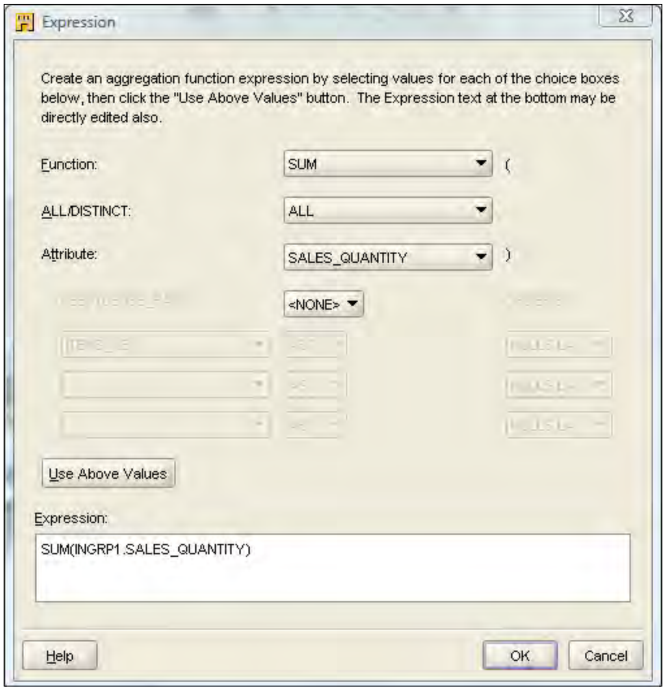
Attribute	Data type	Length	Precision	Scale	Second...	Description
ITEM_LIST_PRICE	NUMBER		6	2		
ITEM_DEPT	VARCHAR2	50				
STORE_NAME	VARCHAR2	50				
STORE_ADDRESS1	VARCHAR2	60				
STORE_ADDRESS2	VARCHAR2	60				
STORE_CITY	VARCHAR2	50				
STORE_STATE	VARCHAR2	50				
STORE_ZIP	VARCHAR2	50				
COUNTRY	VARCHAR2	50				
DATE_SOLD	DATE					
STORE_NUMBER	VARCHAR2	10				
SALES_QUANTITY	NUMBER		0	0		
AMOUNT	NUMBER		10	2		
REGION_NAME	VARCHAR2	50				

We'll click on the **OK** button to close the **AGGREGATOR Editor** dialog box.

11. Now we need to apply the `SUM()` function to these two new attributes, so we'll click on **SALES_QUANTITY** in the **OUTGRP1** group of the Aggregator. In the **Properties** window of the **Mapping Editor** on the left, click on the ellipsis beside the **Expression** property to launch the **Expression** editor for this attribute.

12. We'll immediately notice something different. The **Expression** editor for output attributes of an Aggregator is custom built to apply aggregation functions. We'll select **SUM** from the **Function** drop-down menu, **ALL** from the **ALL/DISTINCT** drop-down menu, and **SALES_QUANTITY** from the **Attribute** drop-down menu. We'll then click on the **Use Above Values** button and the expression will fill in showing the **SUM** function applied to the **SALES_QUANTITY** attribute. This is shown in the next screenshot of the **Expression** editor:

Download at Boykma.Com


Expression

Create an aggregation function expression by selecting values for each of the choice boxes below, then click the "Use Above Values" button. The Expression text at the bottom may be directly edited also.

Function: SUM (

ALL/DISTINCT: ALL

Attribute: SALES_QUANTITY)

<NONE>

Use Above Values

Expression:

SUM(INGRP1.SALES_QUANTITY)

13. We'll click on the **OK** button to save the expression and close the dialog box. Then we'll do the same thing for the `AMOUNT` output attribute of the Aggregator, but will select **AMOUNT** for the **Attribute** drop-down menu.

We're almost done now. We've included:

- The source tables we need to pull the data from
- The target table we're going to store the data in
- A Joiner operator to join together the source tables
- An Aggregator to sum up the data

We have also connected the source tables as input to the Joiner operator and the Joiner as input to the Aggregator operator. The only thing left is to connect the output attributes of the Aggregator operator to the target input attributes. Before doing that, let's make the target table operator box big enough to display all its attributes at once without having to scroll. Just click and hold on the bottom edge of the **POS_TRANS_STAGE** window and drag the window down until all the attributes are visible. Do the same to the **Aggregator operator window**, but make sure only the output group is expanded. The input group should be collapsed because we're finished working with it for now.

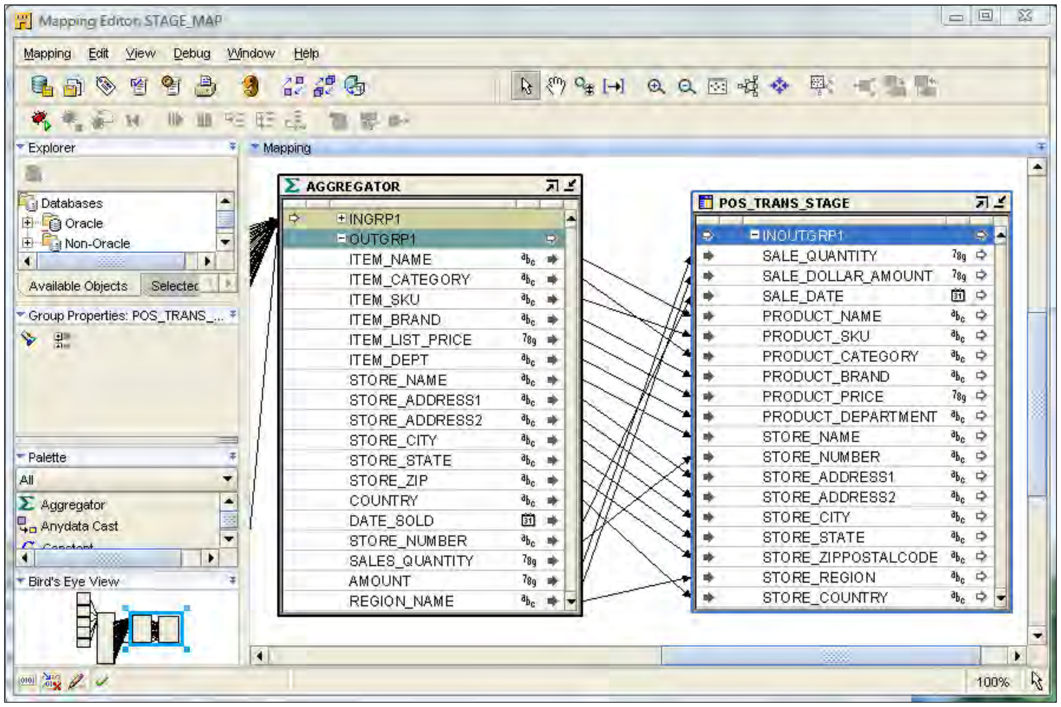
Download at Boykma.Com

Make the following attribute connections between the **Aggregator** and the **POS_TRANS_STAGE** table by clicking and dragging a line between attributes. We'll do individual attributes this time, not the whole group.

- **SALES_QUANTITY** to **SALE_QUANTITY**
- **AMOUNT** to **SALE_DOLLAR_AMOUNT**
- **DATE_SOLD** to **SALE_DATE**
- **ITEM_NAME** to **PRODUCT_NAME**
- **ITEM_SKU** to **PRODUCT_SKU**
- **ITEM_CATEGORY** to **PRODUCT_CATEGORY**
- **ITEM_BRAND** to **PRODUCT_BRAND**
- **ITEM_LIST_PRICE** to **PRODUCT_PRICE**
- **ITEM_DEPT** to **PRODUCT_DEPARTMENT**
- **STORE_NAME** to **STORE_NAME**
- **STORE_NUMBER** to **STORE_NUMBER**
- **STORE_ADDRESS1** to **STORE_ADDRESS1**
- **STORE_ADDRESS2** to **STORE_ADDRESS2**
- **STORE_CITY** to **STORE_CITY**
- **STORE_STATE** to **STORE_STATE**
- **STORE_ZIP** to **STORE_ZIPPOSTALCODE**
- **REGION_NAME** to **STORE_REGION**
- **COUNTRY** to **STORE_COUNTRY**

If we focus on just the **Aggregator** and the **POS_TRANS_STAGE** operators our mapping should now look like the following after making all those connections:

Download at Boykma.Com



Notice that it's not always possible to avoid overlapping lines altogether, but we just want to avoid them as much as we can for readability. In this case, the overlapping lines are a result of different ordering of the attributes in the **AGGREGATOR** operator and the **POS_TRANS_STAGE** table. Changing this would involve recreating the table to reorder columns, and that is just not worth the effort. It will often be the case that sources and targets don't line up like that, but when we add intervening operators over which we have a more direct control, that is where we can focus our efforts on being neat and orderly. Also, your ordering of attributes in the **OUTGRP1** group of the **AGGREGATOR** operator may be different depending on the order in which you mapped the attributes from the Joiner. The order is not important as long as all the required attributes are present.

This completes the staging mapping. We've seen how to create a complete mapping in the Warehouse Builder. We'll make sure to save at this point so that we don't lose anything before we move on.

Download at Boykma.Com

Summary

We saw how to create a mapping in the Warehouse Builder, including how to design a staging area table, build it with the **Data Object Editor**, design a mapping to populate it, and then create the mapping using source and target operators, and the intervening Joiner and Aggregator operators. We also got to use a Transformation Operator.

Now that we have completed our staging table mapping, we need to design mappings to get our dimensions and cube populated from the data in the staging table. We'll do that in the next chapter while taking a look at transformations, which are a key feature of the Warehouse Builder for loading and cleaning up data.

Download at Boykma.Com

7
ETL: Transformations and Other Operators

Now we have completed our first mapping, but there is still more to do. The mapping we completed in the previous chapter was simple, as we just took data from our source database and loaded it into a staging table. We were not at all concerned about what format the data was in, and just wanted to get the data into our staging table in the target environment as quickly as possible. The only other operators we needed besides the source and target tables were a Joiner to pull together the source tables for storing in the staging table, an Aggregator to sum up the data, and a Transformation Operator to truncate the date to remove the time portion. In the previous chapter, we made use of only a very small subset of the Warehouse Builder operators to load and transform data from source into target. We will continue building mappings in this chapter to make use of additional features.

We will be introduced to the concept of transformations and operators that are available in OWB, which can be used for transforming and manipulating data between source and target. We'll do this by building additional mappings for loading data into our `STORE` and `PRODUCT` dimensions, and loading of our `SALES` cube. Along the way, we'll get to build a quick mapping for creating and loading a table that will be used as a lookup table. As we build the mappings, we'll discuss in more detail some of the additional operators we'll need. Thus, we will begin to see the real power and flexibility the Warehouse Builder provides us for loading a data warehouse. When we complete the mappings in this chapter, we will have a complete collection of objects and mappings. We can deploy and run these to build and load our data warehouse.

The building of the mappings in this chapter will be very similar to those in the previous chapter, with the addition of a few more operators. The basic procedure is the same—start with adding a source and a target, and then include any operators in between needed for data flow and transformation. In the last chapter, we were introduced to a couple of data flow operators—the Joiner and the Aggregator. Let's start this chapter with the `STORE` dimension and we'll see some new operators that are involved in transformations.

Download at Boykma.Com

STORE mapping

Let's begin by creating a new mapping called `STORE_MAP`. We'll follow the procedure in the previous chapter to create a new mapping. In the **Design Center**, we will right-click on the **Mappings** node of the **ACME_DW_PROJECT | Databases | Oracle | ACME_DWH** database and select **New...**. Enter **STORE_MAP** for the name of the mapping and we will be presented with a blank **Mapping Editor** window. In this window, we will begin designing our mapping to load data into the `STORE` dimension.

Adding source and target operators

In the last chapter, we loaded data into the `POS_TRANS_STAGE` staging table with the intent to use that data to load our dimensions and cube. We'll now use this `POS_TRANS_STAGE` table as our source table. Let's drag this table onto the mapping from the **Explorer** window. Review the *Adding source tables* section of the previous chapter for a refresher if needed.

The target for this mapping is going to be the `STORE` dimension, so we'll drag this dimension from **Databases | Oracle | ACME_DWH | Dimensions** onto the mapping and drop it to the right of the `POS_TRANS_STAGE` table operator. Remember that we build our mappings from the left to the right, with source on the left and target on the right. We'll be sure to leave some space between the two because we'll be filling that in with some more operators as we proceed.

Now that we have our source and target included, let's take a moment to consider the data elements we're going to need for our target and where to get them from the source. Our target for this mapping, the `STORE` dimension, has the following attributes for the `STORE` level for which we'll need to have source data:

- `NAME`
- `STORE_NUMBER`
- `ADDRESS1`
- `ADDRESS2`
- `CITY`
- `STATE`
- `ZIP_POSTALCODE`
- `COUNTY`
- `REGION_NAME`

Download at Boykma.Com

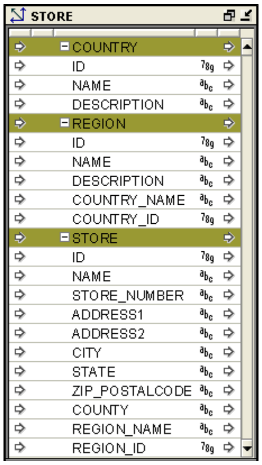
For the REGION level, we'll need data for the following attributes:

- NAME
- DESCRIPTION
- COUNTRY_NAME

For the COUNTRY level, we'll need data for the following attributes:

- NAME
- DESCRIPTION

The complete and fully expanded STORE dimension in our mapping appears like the following screenshot:



We might be tempted to include the ID fields in the above list of data elements for populating, but these are the attributes that will be filled in automatically by the Warehouse Builder. The Warehouse Builder fills them using the sequence that was automatically created for us when we built the dimension. We don't have to be concerned with connecting any source data to them. We discussed that sequence back in *Chapter* 4 when we designed our dimensions.

Now that we know what we need to populate in our STORE dimension, let's turn our attention over to the POS_TRANS_STAGE dimension for the candidate data elements that we can use. In this table, we see the following data elements for populating data in our STORE dimension:

- STORE_NAME
- STORE_NUMBER
- STORE_ADDRESS1
- STORE_ADDRESS2

Download at Boykma.Com

- STORE_CITY
- STORE_STATE
- STORE_ZIPPOSTALCODE
- STORE_REGION
- STORE_COUNTRY

It is easy to see which of these attributes will be used to map data to attributes in the STORE level of the STORE dimension. They will map into the corresponding attributes in the dimension in the **STORE** group. We'll need to connect the following attributes together:

- STORE_NAME to NAME
- STORE_NUMBER to STORE_NUMBER
- STORE_ADDRESS1 to ADDRESS1
- STORE_ADDRESS2 to ADDRESS2
- STORE_CITY to CITY
- STORE_STATE to STATE
- STORE_ZIPPOSTALCODE to ZIP_POSTALCODE
- STORE_REGION to REGION_NAME

There is another attribute in our STORE dimension that we haven't accounted for yet—the COUNTY attribute. We don't have an input attribute to provide direct information about it. It is a special case that we will handle after we take care of these more straightforward attributes and will involve the lookup table that we discussed earlier in the introduction of this chapter.

We're not going to directly connect the attributes mentioned in the list by just dragging a line between each of them. There are some issues with the source data that we are going to have to account for in our mapping. Connecting the attributes directly like that would mean the data would be loaded into the dimension as is, but we have investigated the source data and discovered that much of the source data contains trailing blanks due to the way the transactional system stores it. Some of the fields should be made all uppercase for consistency.

Given this additional information, we'll summarize the issues with each of the fields that need to be corrected before loading into the target and then we'll see how to implement the necessary transformations in the mapping to correct them:

- STORE_NAME, STORE_NUMBER: We need to trim spaces and change these attributes to uppercase to facilitate queries as they are part of the business identifier

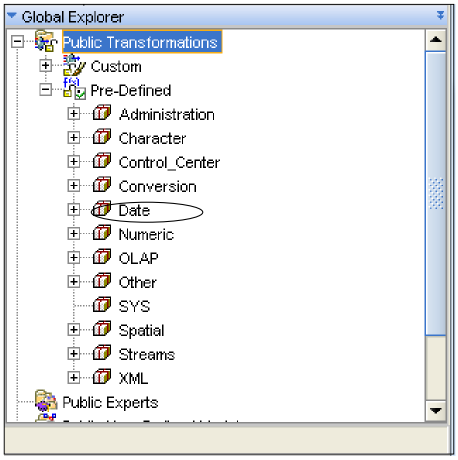
Download at Boykma.Com

- STORE_ADDRESS1, ADDRESS2, CITY, STATE, and ZIP_POSTALCODE: We need to trim spaces and change the STATE attribute to uppercase
- STORE_REGION: We need to trim spaces and change this attribute to uppercase

All of these needs can be satisfied and we can have the desired effect by applying pre-existing SQL functions to the data via Transformation Operators.

Adding Transformation Operators

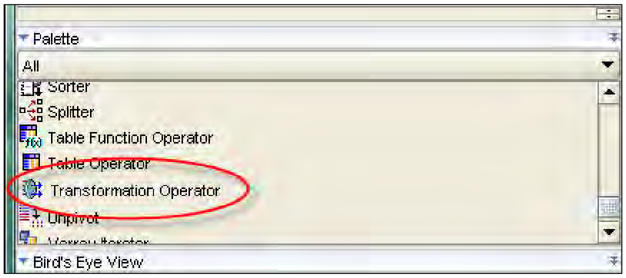
The Transformation Operator is a generic operator that is used to represent several built-in or custom-built functions or procedures for operating on data in order to make some kind of change or transformation to it. Let's take a look at the available list of transformations. In the **Design Center**, we can look at a list of available transformations either custom or pre-built in the database in the **Global Explorer** panel under **Public Transformations**. There are several categories of transformations available to us as shown in the following screenshot:



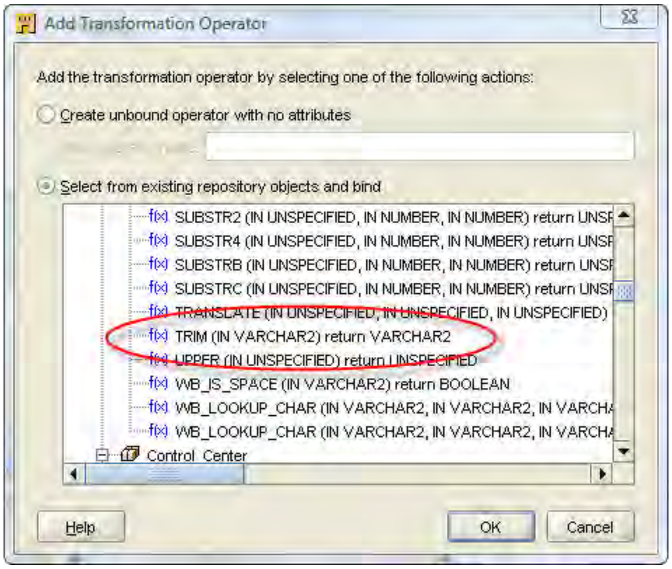
We are primarily interested in the **Character** category because that is where we'll find functions that operate on character strings, and that can convert them to uppercase and remove whitespace. We can expand any of these lists to take a look at the names of the various transformations names available. We can also go to the online help for detailed explanations of all the functions in the **Transformations** heading under the **ETL Design Reference** section of the online help table of contents. You can access this by selecting **Help | Table of Contents** from the main menu of the Design Center, or by pressing the *F1* key. The particular transformation names we need under the character heading are the `upper()` function to convert to uppercase and the `trim()` function to remove whitespace.

Download at Boykma.Com

We can now move back to the Mapping Editor where we're creating our `STORE_MAP` mapping and begin to add the transformations, and through them connect source to target. The first data element we'll map is the store name, so let's drag a Transformation Operator onto the mapping and drop it between the `POS_TRANS_STAGE` table and the `STORE` dimension. We can find the **Transformation Operator** in the **Palette** window as shown in the following screenshot:



After dropping the **Transformation Operator** on the mapping, it will pop up a dialog box where we select the transformation we want to use. We have two options in this dialog box—create an unbound operator (basically, one that is not tied to an existing repository object) or select from an existing object. We'll select an existing one because we know that the function that will suit our purpose already exists. We'll scroll the window down until we see the **TRIM()** function as shown in the following screenshot:


Add Transformation Operator

Add the transformation operator by selecting one of the following actions:

Create unbound operator with no attributes

Select from existing repository objects and bind

SUBSTR2 (IN UNSPECIFIED, IN NUMBER, IN NUMBER) return UNSP
SUBSTR4 (IN UNSPECIFIED, IN NUMBER, IN NUMBER) return UNSP
SUBSTRB (IN UNSPECIFIED, IN NUMBER, IN NUMBER) return UNSP
SUBSTRC (IN UNSPECIFIED, IN NUMBER, IN NUMBER) return UNSP
TRANSLATE (IN UNSPECIFIED, IN UNSPECIFIED, IN UNSPECIFIED)
TRIM (IN VARCHAR2) return VARCHAR2
UPPER (IN UNSPECIFIED) return UNSPECIFIED
WB_IS_SPACE (IN VARCHAR2) return BOOLEAN
WB_LOOKUP_CHAR (IN VARCHAR2, IN VARCHAR2, IN VARCHA
WB_LOOKUP_CHAR (IN VARCHAR2, IN VARCHAR2, IN VARCHA
Control Center

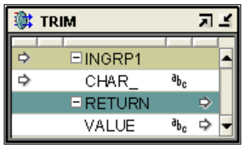
Download at Boykma.Com

Searching for a function

If we want to find the function quickly, rather than manually scrolling the window down, there's a not-so-obvious feature of this dialog box called the search capability. If we start typing the name of the function we want, it will automatically scroll down the list with each letter typed, until it settles on the one we want. For example, type a *T* and it highlights the very first line, **TRANSFORMATIONS**. Type an *R* next and it stays on that line because transformations start with those two letters. But type an *I* next and it jumps right down to the **TRIM** function we need. This option is much better to quickly find what we're looking for than manually scrolling the window. This option is a great help as it's so easy to scroll right by what we're looking for without realizing it.

If you click anywhere on the window before typing, the search string will start the search at that point.

We'll now click on the **TRIM** function in the window and then on the **OK** button. This will display a **TRIM** Transformation Operator window on our mapping. It is like any other operator in that it has attributes, which are in groups depending on whether they are input, output, or both. In this case, a **TRIM** operator has one input attribute and one output attribute. The input attribute is the string we want to trim the whitespace from and the output attribute represents the result of applying the **TRIM** operator to the input string. It looks like the following screenshot:



With all of these Transformation Operators that we can select from, the attribute names will appear similar to the above character attributes named `CHAR_`, a return value named `VALUE`. We can change these if we want, but this will become tedious when large numbers of Transformation Operators are required. Leaving the Transformation Operators's attributes as they are will not affect the operation of the mapping.

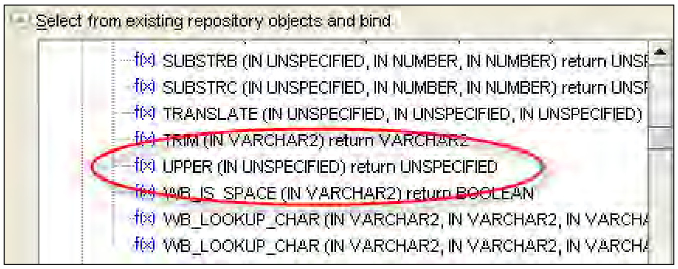
Download at Boykma.Com

We can now connect our `STORE_NAME` attribute in the `POS_TRANS_STAGE` mapping table operator to this new **TRIM** operator. We'll drag a line from `STORE_NAME` to **CHAR_** in the **TRIM** operator. This succeeds in mapping the input for our new **TRIM** Transformation Operator, but now we need to map the output somewhere. We could just drag a line from the **VALUE** output attribute over to the **STORE** dimension. But we've said before that we need to apply an **UPPER** transformation on this value as well as a trim, so the value that ultimately gets loaded into our dimension will be in all uppercase letters.

Upper and lowercase issues

When working between an MS SQL Server Database and an Oracle Database, we will frequently find that the case of the strings we're working with becomes an issue. The Oracle Database is very case sensitive. If we store a string in the database as '`Some String`', then searching for '`some string`' will not get a match. It will match in SQL Server, even though the case is different. This is why, it is a good idea to store key fields that uniquely identify a record in the database in all uppercase. By doing so, we won't encounter a possible situation where two records get loaded into our data warehouse that differ by only the case of the key identifier, or we don't get a match at all because of a different case.

Now we need an **UPPER** transformation added for our `STORE_NAME`, so let's drag another Transformation Operator onto the mapping and drop it to the right of the **TRIM** operator. It is perfectly acceptable and very common, in fact, to have to map the output from one Transformation Operator into the input of another Transformation Operator. We will select the **UPPER()** function this time from the resulting pop-up window. It is close to the **TRIM** function in the dialog box as shown in the following screenshot:



Select from existing repository objects and bind

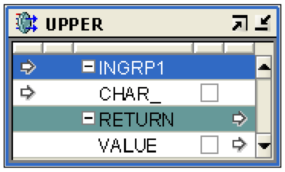
```
SUBSTRB (IN UNSPECIFIED, IN NUMBER, IN NUMBER) return UNSP
SUBSTRC (IN UNSPECIFIED, IN NUMBER, IN NUMBER) return UNSP
TRANSLATE (IN UNSPECIFIED, IN UNSPECIFIED, IN UNSPECIFIED)
TRIM (IN VARCHAR2) return VARCHAR2
UPPER (IN UNSPECIFIED) return UNSPECIFIED
WB_IS_SPACE (IN VARCHAR2) return BOOLEAN
WB_LOOKUP_CHAR (IN VARCHAR2, IN VARCHAR2, IN VARCHA
WB_LOOKUP_CHAR (IN VARCHAR2, IN VARCHAR2, IN VARCHA
```

Download at Boykma.Com

The **UPPER()** function is similar to the **TRIM()** function in the number of arguments it takes and the value it returns—which is one in both the cases. It is different in that the **UPPER()** function does not specify the type of the arguments as the **TRIM()** function does. We can see from the previous screenshot that the type is listed as **UNSPECIFIED**. We know these are character functions because that is where we found them in the list. As a **TRIM** function removes characters (blank characters) from a string, this string must be a `varchar2` string and not a string of a `char` type. A `varchar2` string is a variable length string up to the maximum length it was defined with; so if you remove some characters from it, it just shrinks in size. However a `char` string is a string with a fixed length.

The database will fill up a char string with blanks up to the maximum size of the string if you store a string that is smaller than the defined size of the `char` string. A **TRIM()** will have no effect on this kind of field. An **UPPER()** function, on the other hand, will work on a string of any type. The *Oracle Database SQL Language Reference* manual (which can be found at `http://download.oracle.com/docs/cd/B28359_01/server.111/b28286/functions214.htm#i90176`) indicates that the parameter can be any of the following: `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. When we look up the **TRIM()** function, we see that it can only be a **VARCHAR2** for input.

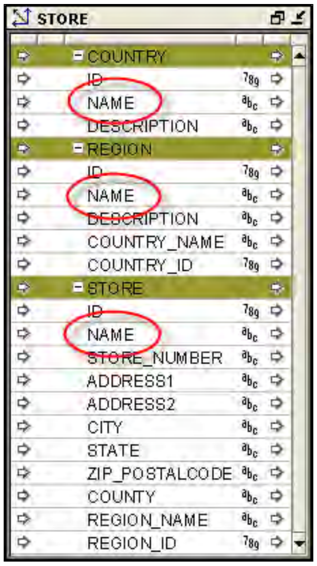
Thus, the **UPPER** transformation is not able to know beforehand the exact type of the input and output parameters. It will know about that only when we drag an actual value to it. There is also a difference in how the operator looks on the mapping just after being dropped. We can see in the following screenshot that unlike other operators in our mapping, the attribute type indicators in this operator (that appear to the right of the attributes) show as blank boxes:



These blank boxes will fill in automatically at some point after an attribute gets mapped to it, so let's continue. We will need the output of the **TRIM** operator to be the input of this one, so we'll drag a line from `VALUE` in the **TRIM** operator to `CHAR_` in the **UPPER** operator. The type indicator in the window on the mapping will eventually update automatically to reflect the type that was connected to it, which is `VARCHAR2` in this case.

Download at Boykma.Com

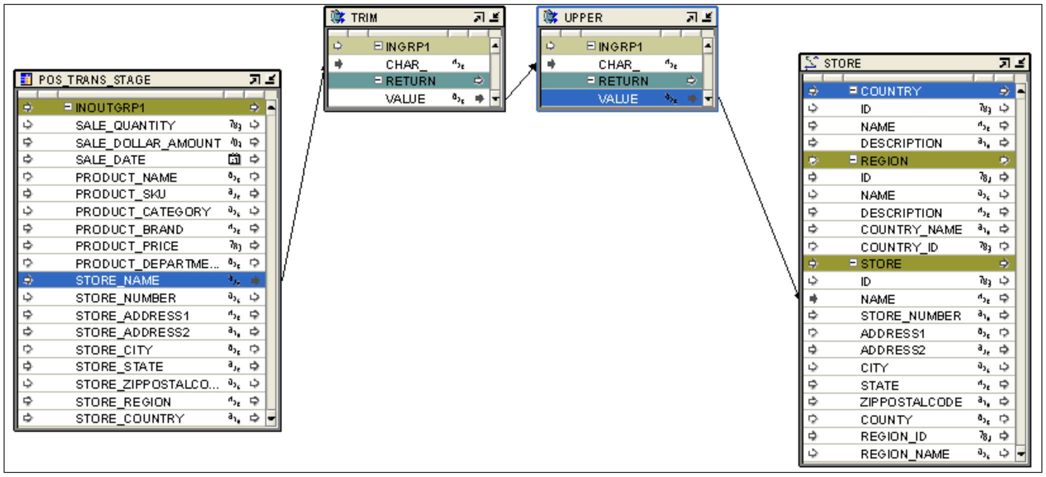
We can now connect to the STORE dimension as we don't have any more transformations that we need to do to the STORE_NAME, but we need to decide where in the STORE dimension to connect. This is where it could be easy to make the wrong connection because looking at the **STORE** dimension we see that it has three **NAME** attributes, all of which can be used as input, as circled in the following screenshot:



Here the key in deciding which attribute to use is to recall our discussion of the design of our dimension back in *Chapter 4*. There we talked about the levels and hierarchy that can exist in a dimension, and how certain attributes can be designated as **dimension attributes**, which can be found at every level of the dimension. In this case, the **NAME** attribute is just such an attribute. The hierarchy in this case is COUNTRY, REGION, STORE; and each level of the hierarchy has a NAME associated with it. This is the name of the store, so let's make sure to drag the line to the **NAME** attribute in the **STORE** group of the **STORE** dimension operator, which is the bottommost operator in the above screenshot.

At this point, our mapping should look roughly similar to the following. The placement of the operators on the mapping will vary, but it should generally be similar to the **POS_TRANS_STAGE** mapping table on the left as input, the **TRIM** and **UPPER** operators in the middle connecting the **STORE_NAME** attribute from input to the **NAME** attribute of the **STORE** group in the **STORE** dimension as output:

Download at Boykma.Com



To keep our mapping from becoming too cluttered, collapse the transformation windows into their icon view after making all the connections through them. The icons take up much less space on the mapping and can always be opened again by double-clicking on them if needed in the future. We can click on the down arrow in the upper-right corner of the window to collapse them into the icon view.

Next, we'll take care of the `STORE_NUMBER` as it is the second part of the business identifier for the store. The name and number of the store are what uniquely identify a single store in the ACME Toys and Gizmos company, and are handled similarly in our mapping. The same two transformations are needed for the `STORE_NUMBER` field as for `STORE_NAME` from the `POS_TRANS_STAGE` input table, but we can't reuse the existing two transformations we just dropped onto our mapping. We will need to drag two more transformations to our mapping, making one a **TRIM** and another an **UPPER** just as we did for the `STORE_NAME`. We'll connect them in a manner similar to how we connected the previous two transformations, but this time we'll start with the `POS_TRANS_STAGE` mapping table operator. We will connect the `STORE_NUMBER` attribute to the input of the **TRIM**, the output of the **TRIM** to the input of the new **UPPER** we just dropped onto the mapping, and the output of the **UPPER** to the `STORE_NUMBER` attribute of the `STORE` dimension. There is only one `STORE_NUMBER` attribute in the dimension, unlike the name, because the `STORE_NUMBER` is not defined as a dimension attribute; it exists only at the `STORE` level as a level attribute.

Download at Boykma.Com

At this point we have our `STORE_NAME` and `STORE_NUMBER` attributes connected to the dimension, and we'll continue with the two address fields, the city, the state, and the zip/postal code field. We determined that these fields will need to have spaces trimmed, but we do not want to make them uppercase except for the state field. They are not a part of the unique business identifier for an individual store and, apart from the state field, can be any combination of characters and/or numbers, which make them less likely to be queried for. The state field contains states in the US, which are commonly expressed as two uppercase characters, and so we'll apply the **UPPER** transformation to it.

We will need six more Transformation Operators dropped into our mapping, with five being for **TRIM** operators for each of those five fields and one for an `UPPER()` function to use for the state field. The following attributes of the `POS_TRANS_STAGE` mapping table operator will provide the input for the five **TRIM** operators:

- `STORE_ADDRESS1`
- `STORE_ADDRESS2`
- `STORE_CITY`
- `STORE_STATE`
- `STORE_ZIPPOSTALCODE`

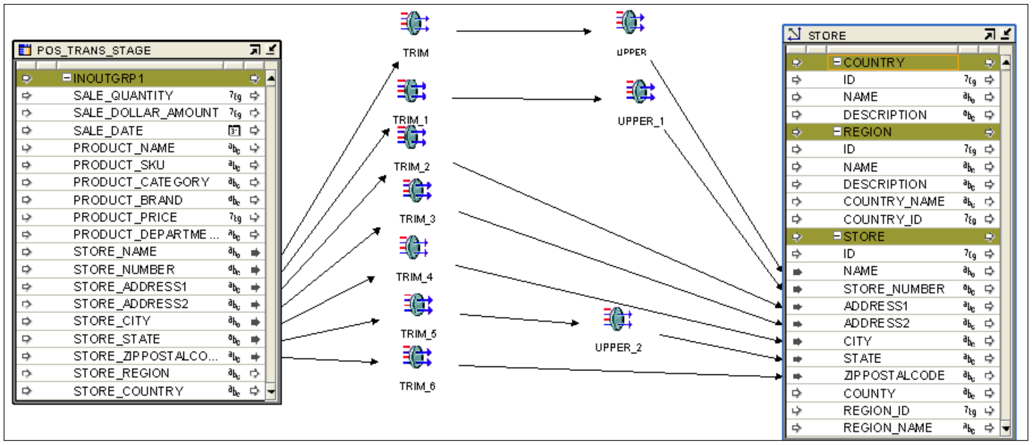
The output of the **TRIM** operators for all but the `STORE_STATE` attribute will be connected directly to **STORE** level attributes of the **STORE** dimension as follows:

- `ADDRESS1`
- `ADDRESS2`
- `CITY`
- `ZIP_POSTALCODE`

The **TRIM** output for the `STORE_STATE` attribute will be connected to the **UPPER** Transformation Operator, and the output from the **UPPER** operator will be connected to the **STATE** attribute in the **STORE** dimension.

After making all these connections, our mapping should now look similar to the following with all the Transformation Operators collapsed into their icon views:

Download at Boykma.Com



We're not done with this mapping yet as we still have to map the STORE_REGION attribute to the **STORE** level, and map both the **REGION** and **COUNTRY** levels. Before continuing, let's save our work so far with the **Mapping | Save All** menu entry on the **Mapping Editor**. We can also use the *Ctrl+S* key combination.

Information about both the region and country comes from two attributes in our source staging table, the STORE_REGION and STORE_COUNTRY attributes. These are character fields for the name of the region and country the store is located in. When we designed our STORE dimension in *Chapter 4*, a NAME and DESCRIPTION field were created for us by default. We decided to leave it that way as that is a common design technique for dimensions and avoids the error we mentioned back then about not having any updatable fields. As we don't have separate name and description fields to draw from at this point, we'll just fill the same information into both fields in the STORE dimension. The NAME field is identified as the business identifier, so we'll put the value we store there into uppercase and leave the description in whatever case the source was in.

Let's start with the region attribute. We can see in our STORE dimension that there is a REGION_NAME attribute in the **STORE** group (level). This attribute indicates in which region on the **REGION** level this store is located. Looking at the **REGION** level we can see that there is a **COUNTRY_NAME** located there, which indicates the country from the **COUNTRY** level where the region is located. In terms of our mapping, this determines where we map the STORE_REGION and STORE_COUNTRY attributes to.

The first mapping change we'll do for the region is to finish up the **STORE** level attributes by mapping the STORE_REGION from the stage table to the REGION_NAME attribute in the STORE dimension, **STORE** level. We indicated earlier that names should be capitalized and spaces trimmed, so we'll drag two more Transformation Operators into our mapping—**TRIM** and **UPPER**—and map the STORE_REGION to the **TRIM**, the **TRIM** to the **UPPER**, and the **UPPER** to the REGION_NAME field.

Download at Boykma.Com

This completes the **STORE** level except for the COUNTY attribute, and we still have this attribute plus the **REGION** and **COUNTRY** levels to complete. At this point, we've become more proficient in doing our mapping and including transformations. So we'll just continue to the **REGION** level and add the following connections and transformations without having to walk through each one in detail:

- STORE_REGION to NAME in the **REGION** level using **TRIM** and **UPPER** transformations
- STORE_REGION to DESCRIPTION in the **REGION** level using a **TRIM** transformation
- STORE_COUNTRY to COUNTRY_NAME in the **REGION** level using **TRIM** and **UPPER** transformations
- STORE_COUNTRY to NAME in the **COUNTRY** level using **TRIM** and **UPPER** transformations
- STORE_COUNTRY to DESCRIPTION in the **COUNTRY** level using a **TRIM** transformation

But we'll want to implement the following tip to make our mapping easier and less cluttered.

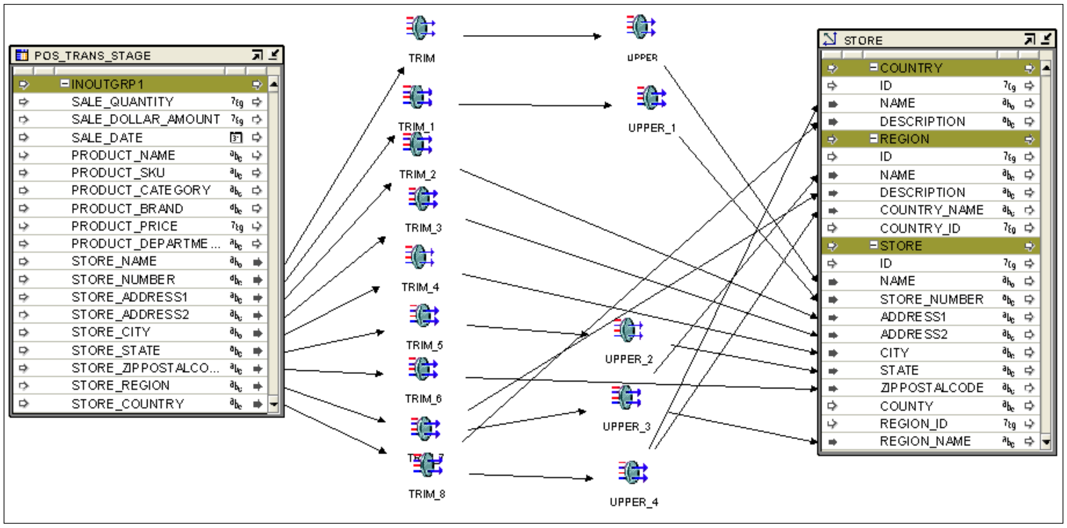
We have had a couple of instances earlier where the same input attribute needs to be mapped to more than one target attribute. We learned previously that we couldn't reuse a Transformation Operator on two different input attributes. However, we can reuse a Transformation Operator if the output goes to two different attributes. Multiple connections can be created from an output attribute in an operator, but only one input connection is allowed.

For example, we can use just one **TRIM** operator on the REGION_NAME and have its output go to an **UPPER** operator and also directly to the DESCRIPTION attribute in the **REGION** group of the **STORE** dimension. The output of the **UPPER** operator can then be connected to both the REGION_NAME of the **STORE** level and the NAME attribute of the **REGION** level. The same technique can be applied to the mappings for the COUNTRY_NAME.

The bottom line is to reuse the **TRIM** and **UPPER** operators just added for the NAME in the **REGION** level, and add one **TRIM** and one **UPPER** operator for the COUNTRY_NAME.

After adding the two additional transformations and making the connections already mentioned, our mapping should now look similar to the following screenshot:

Download at Boykma.Com



Our **STORE** dimension is now mapped for every attribute except for the `COUNTY` attribute. We've saved this one for last because it is the most complex of our attributes to map for this dimension. The reason is that we don't have an exact match with an attribute from our input staging table to use. Let's save our work at this point and then investigate further how we need to map this attribute.

Using a Key Lookup operator

Key Lookup operators, as the name implies, are used for looking up information from other sources based on some key attribute(s) in a mapping. This is exactly what we will need to do to get the information for the `COUNTY` attribute of our **STORE** dimension. However, only tables, views, dimensions, and cubes can be used as the source for this operator. This means we need a table that can be used to look up the required county information. Back in *Chapter 2*, we imported the source metadata for a flat file called `counties.csv`, creating a file in our `ACME_FILES` module in Design Center for this file that contains the names of counties. It looks like we ought to be able to use the information in that file to build a lookup table, so that's exactly what we're going to do right now. We will use a simple Warehouse Builder mapping to do it in a couple of easy steps. First, we will need to create an external table to represent the `counties.csv` file. We could use the `counties.csv` file directly, but as we discussed back in *Chapter 2* that would require using the SQL*Loader utility, which would not be consistent with the PL/SQL access that can be used for all the other sources. So we will create an external table using the simple steps outlined in the next section, and then follow that by using that external table as the source in a new mapping to load a lookup table.

Download at Boykma.Com

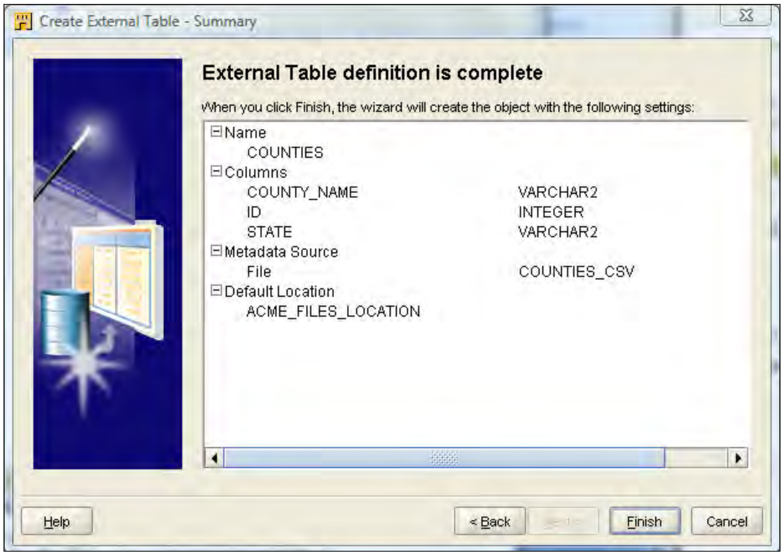
Creating an external table

In *Chapter 2*, we imported metadata from the `counties.csv` file which was created in a module separate from our main database module because a file is not a part of the database. However, external tables are created in the database as they are accessed just like regular database tables. However, unlike a regular table whose data is stored in the database, an external table's data is stored in a flat file that is external to the database.

External tables are created under the **Oracle | ACME_DWH | External Tables** node in the **Design Center**, so we'll right-click on it and select **New...** from the pop-up menu. This will launch the **External Table** wizard, which will guide us through the process. It is a three-step process that involves providing a name to use for the external table, specifying the file to use, and specifying the default location. The steps are as follows:

1. By clicking on **Next** on the **Welcome** screen, we come to the screen labeled **Step 1**. We'll name this external table **COUNTIES** and click on **Next** to continue to the screen labeled **Step 2**.
2. In this step we'll select the file that contains the metadata for the external table. It will display the name of any files that have been defined in our **Files** module. We can see our **COUNTIES_CSV** file listed, so we'll select that and click on **Next** to continue.
3. This brings us to the screen labeled **Step 3** where we will select the default location to use for this table. The drop-down menu on this screen will display the file locations that have been defined in the Design Center. We will select the **ACME_FILES_LOCATION** entry, which is for the files that exist for this project. Clicking on **Next** will bring us to the **Summary** screen where we can verify the information we just specified. It should look similar to the following screenshot:

Download at Boykma.Com



External Table definition is complete

When you click Finish, the wizard will create the object with the following settings:

```
⊟Name
    COUNTIES
⊟Columns
    COUNTY_NAME                 VARCHAR2
    ID                          INTEGER
    STATE                       VARCHAR2
⊟Metadata Source
    File                        COUNTIES_CSV
⊟Default Location
    ACME_FILES_LOCATION
```

If we see anything we'd like to change, we can click on the **Back** button to move back through the screens to make any changes and click on **Next** until we get back here.

4. When we click on the **Finish** button, it will create a new entry called **COUNTIES** under the **External Tables** node in our project in the Design Center.

The wizard has created an external table with the column attributes that were listed in the **Summary** screen. These attributes correspond to the fields that are stored in the `counties.csv` flat file. We can query this table just as we query a table in the database.

Creating and loading a lookup table

Now that we have our source table defined for our new lookup table, let's create a new mapping called `COUNTIES_LOOKUP_MAP` using the same method we've used previously. The steps to create a lookup table are:

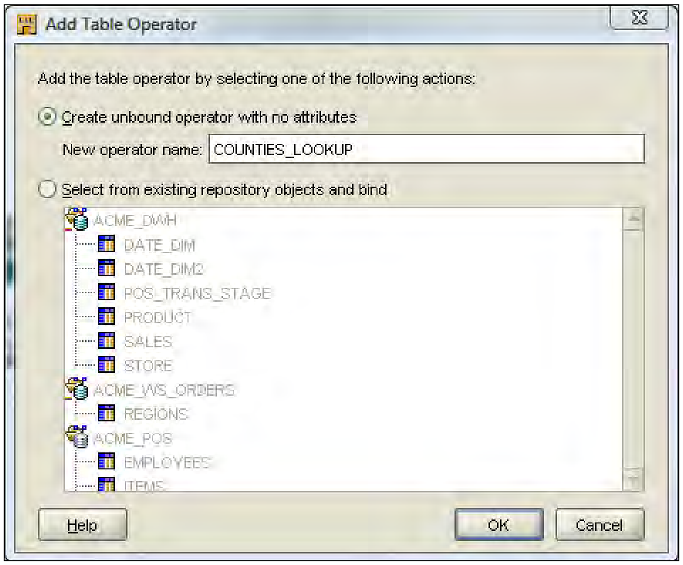
1. Right-click on the **Mappings** node, select **New...**, enter **COUNTIES_LOOKUP_MAP** in the name field, and click on the **OK** button.
2. In the **Mapping Editor** that pops up, let's drag an **External Table Operator** from the **Palette** window onto the mapping.

Download at Boykma.Com

3. On the **Add External Table Operator** pop-up window that appears, our **COUNTIES** external table is visible. We will select that and click on the **OK** button to continue. This will drop an External Table Operator on our mapping that is bound to our **COUNTIES** external table.
4. We need to get that data loaded into a regular table in the database, so next we'll drag a **Table Operator** onto the mapping.

As this table doesn't yet exist, there are a few different ways we can go about creating it to hold our county information. We could create the table in the Design Center in the **Tables** node under our database module, and then drag that table into our mapping. Alternatively, we could create the table in the database and then import the metadata for that table as we imported source metadata back in *Chapter 2*, or we can take the path we're taking now to make full use of the Warehouse Builder's automation and flexibility and create the table as we need it.

5. In the resulting **Add Table Operator** pop up that appears, we specify what table we want to add. We've seen this add operator dialog box before, but we've always been choosing an existing object to add. This time we're going to check the first option to **Create unbound operator with no attributes** and we'll give it the name **COUNTIES_LOOKUP** by typing that name into the box. This is shown in the following screenshot:


Add Table Operator

Add the table operator by selecting one of the following actions:

[✓ Create unbound operator with no attributes]

New operator name: COUNTIES_LOOKUP

Select from existing repository objects and bind

- ACME_DWH
  - DATE_DIM
  - DATE_DIM2
  - POS_TRANS_STAGE
  - PRODUCT
  - SALES
  - STORE
- ACME_WS_ORDERS
  - REGIONS
- ACME_POS
  - EMPLOYEES
  - ITEMS

6. We'll click on **OK** and it will drop a Table Operator onto our mapping with no attributes defined in it.

Download at Boykma.Com

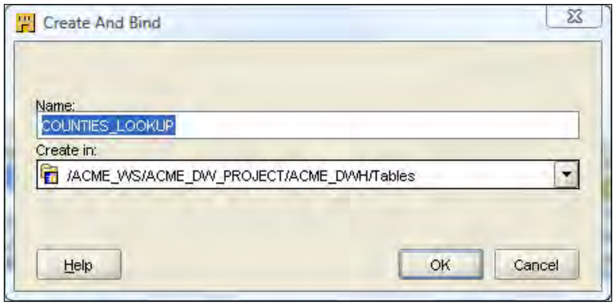
We need to define the attributes and we know we need the data loaded from the external table, so let's use these attributes in our example . We might think we have to enter each of these attributes individually into the Table Operator and then drag a line from the corresponding attribute in the external table over to the new table. But the Warehouse Builder makes this very easy; with one drag we can map an attribute group instead of individual attributes.

Let's drag a line from the output group (`OUTGRP1`) of our `COUNTIES` external table over to the input/output group (`INOUTGRP1`) of our new `COUNTIES_LOOKUP` table. With that one action, the new table operator immediately goes from being empty to having three attributes defined in it. These attributes have names that are the same as the external table attribute names and connecting lines are drawn for all three attributes to map them from the external table. This is very neat, and it just saved us a bunch of time.

This mapping is done. However, there is one more step we need to take to actually create the lookup table definition. Remember we created our table operator as an unbound operator, which means it's not associated with any database object. If we look in the **ACME_DWH | Tables** node, there is no table named `COUNTIES_LOOKUP`. The steps to create a new table object and to bind this operator to it are as follows:

1. When we right-click on the unbounded operator, the pop-up menu has a menu selection called **Create and Bind...**. With this option we will create a new table object in the OWB **Tables** node and bind this operator to it.

   Let's select that menu entry from the pop-up menu and it will present us with the following dialog box:


Create And Bind

Name: COUNTIES_LOOKUP

Create in: /ACME_WS/ACME_DW_PROJECT/ACME_DWH/Tables

2. The name is the same as what we gave to the operator. We could name the underlying bound table something different, but it's best to leave it with the same name for clarity.

Download at Boykma.Com

3. The **Create in:** text field is to specify the module in which to create the new table under our project in the **Design Center**. It has defaulted to the **Tables** node under the current `ACME_DWH` module, and that is exactly where we want it. The drop-down option provides a listing of every **Tables** node in all the modules that are currently defined in our current project if we want to create it in one of those other modules.
4. When we click on the **OK** button on this dialog box, a table is created in the **Tables** node and is bound to the operator.

To verify that, we can navigate to the **ACME_DWH | Tables** node under our database module and there is the new **COUNTIES_LOOKUP** table now. This completes the mapping and table creation. Our new table is now ready to include in a mapping as a Key Lookup operator.

[To ensure that we don't end up with duplicate records in our new lookup table, we can take an extra step to define a primary key on this table.]

When we use the option to create a table in this manner, it creates a basic, no-frills table with no constraints defined on it. To add a primary key, we'll perform the following steps:

1. In the **Design Center**, open the **COUNTIES_LOOKUP** table in the **Data Object Editor** by double-clicking on it under the **Tables** node.
2. Click on the **Constraints** tab.
3. Click on the **Add Constraint** button.
4. Type **PK_COUNTIES_LOOKUP** (or any other naming convention we might choose) in the **Name** column.
5. In the **Type** column, click on the drop-down menu and select **Primary Key**.
6. Click on the **Local Columns** column, and then click on the **Add Local Column** button.
7. Click on the drop-down menu that appears and select the **ID** column.
8. Close the **Data Object Editor**.

Download at Boykma.Com

The new table and mapping is now complete. It is very basic, but gives us an idea of the power of the Warehouse Builder to make our data warehouse design job easier. The mapping just handles inserts into the lookup table from the external table. We could add more bells and whistles to our lookup to handle updates or changes to the existing rows, but that is for more advanced topics.

We'll save our work up to this point with the *Ctrl+S* key combination, and then move on to make use of this new lookup table to retrieve the county information.

Retrieving the key to use for a Lookup Operator

We now have a table definition created and a mapping completed that can load the table to use to look up the county name. But we need a key that will uniquely identify a record in the table and with which we can look up a county. The key has to be a data element that is unique in the file, and it would be the ID column we defined as the primary key for the table. It is a number that does not repeat itself for any of the rows in the file; so given a particular value of that number, we can find the county and the state that the county is in.

We recall from our analysis and importing of source metadata back in *Chapter 2* that the `STORE_NUMBER` data element contained in the `STORES` source table has a code that indicates the county the store is located in for stores in the USA. This is actually a fixed known format, and the positions three through six of the number are actually the code for the location of the store in the county. This number is actually the ID number found in the `counties.csv` flat file, and which is the `ID` in the lookup table. So, we now have a key value that we can use to look up the county. However, there are still some more issues we have to work out before we can use it.

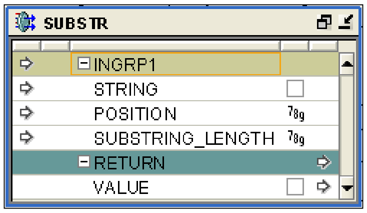
The county `ID` is only a portion of the entire `STORE_NUMBER` field, so we can't just use the `STORE_NUMBER` from input as the direct key to a Key Lookup Operator. We will have to extract the ID number out of it and then convert it to a number before we can use it to look up the county. This implies that some more transformations will be needed, so let's work on getting that county ID extracted from the `STORE_NUMBER` field.

Adding a SUBSTR Transformation Operator

The Transformation Operators available to us in OWB include a `SUBSTR` (or substring) transformation that will do exactly what we need to extract the county ID value out of the `STORE_NUMBER` field. The `SUBSTR` transformation takes three parameters—the string we want to extract the substring from, a number indicating the start position of the substring within the string, and a number indicating the length of the substring to extract.

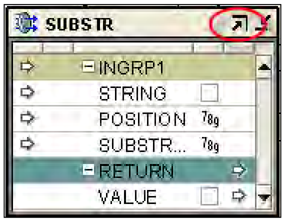
Download at Boykma.Com

So, let's drag a Transformation Operator onto the STORE_MAP mapping between the POS_TRANS_STAGE table and the STORE dimension below all the other Transformation Operators. On the resulting **Add Transformation Operator** pop-up window, select the **SUBSTR()** transformation and it will place the following operator into our mapping:



For the **SUBSTR** operator, we need to make sure we select the correct version as there are five different variants of SUBSTR we could choose from. They are SUBSTR, SUBSTR2, SUBSTR4, SUBSTRB, and SUBSTRC. The main SUBSTR version is the one we want because it works on regular character strings. The others only vary in the type of input character string they operate on. A more in-depth description of the SUBSTR() function and its variants is in the *Oracle Database SQL Language Reference Manual*, which is available online at the **Oracle Database Documentation** web site (http://www.oracle.com/technology/documentation/database.html).

When first dropped on the mapping, this operator may not look exactly like the above screenshot in which the operator is fully expanded. To see the whole operator contents at once, we can click and drag an edge to manually make the window bigger or click on the symbol in the upper-right corner with the arrow pointing upwards as indicated in the following screenshot, which shows the operator before being expanded fully:



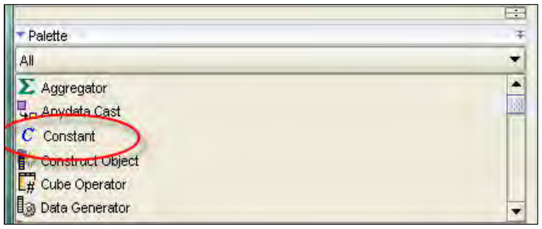
We didn't have this issue with any of the Transformation Operators we included earlier, but it's helpful here for being able to see the entire contents of the operator.

Download at Boykma.Com

All windows on the mapping, and not just the Transformation Operators, have this feature for expanding the window size. We'll find that the table, dimension, and cube operators need to be expanded frequently to see the entire contents, and this is a quick way to do it.

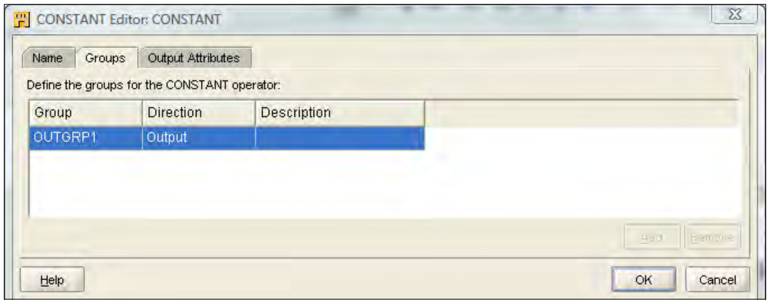
Let's continue mapping attributes to the **SUBSTR** operator. The **STRING** attribute is easy, which will be the `STORE_NUMBER` from the `POS_TRANS_STAGE` table. So let's drag a line from `STORE_NUMBER` down to **STRING**. The position and length are not so obvious. We don't have any values in the source table to use for those two so we need to create something to use.

The second and the third parameter—the position and length—these need to be constant integer values that we supply. By looking at the list of operators available to us in the **Palette** window, we see that there is a **CONSTANT** operator as shown in the following image. We can use this operator by dragging it from the **Palette** window in the **Mapping Editor**:



Adding a Constant operator

We'll click and drag a **Constant** operator onto the mapping to the left of the **SUBSTR** Transformation Operator. We can see that it has an output group called **OUTGRP1** by default. We'll right-click on it and select **Open Details...** from the pop-up menu. This opens an editor on the **CONSTANT** operator, which should look like the following screenshot:



Define the groups for the CONSTANT operator:

Group	Direction	Description
OUTGRP1	Output	

Download at Boykma.Com

The tab that is highlighted when the dialog box opens depends on what was right-clicked. As we can see, this editor has tabs for editing the **Name** of the operator, the **Groups**, and the **Output Attributes** of the output group. The **Constant** operator only allows output, so there is no input group defined or allowed. If it was an operator that allowed input, such as a function or procedure that took parameters (for example, the **SUBSTR** operator), there would be an additional tab for **Input Attributes** also.

Clicking on the **Output Attributes** tab we see that there are no attributes currently existing for this operator. This is where we will add our constants that we need for the **SUBSTR** operator. We can actually enter more than one constant in the same operator (which is a good thing to do if we are using those constants together anyway), which we are doing in this case. We could just as easily drag another **Constant** operator onto the mapping for the other constant we need; it's really just a matter of preference. Functionally, the result will be the same when OWB deploys and executes the mapping.

To add an attribute, click on the **Add** button. This will create an output attribute in the group with a default name and data type that we can then edit to suit our purposes. We'll change the name of this first constant attribute to reflect the destination for this value, that is the position attribute of the **SUBSTR** operator, so we'll name it **POSITION** also. Click on the default name, **OUTPUT1**, and it will highlight the name and we can then type in what we want it to be. We'll change the name to **position**.

We don't have to worry about capitalization as the Warehouse Builder will automatically convert everything to uppercase anyway. We'll see this when we click elsewhere on the dialog box and the focus moves out of that field, or if we close the dialog box and then look at the name in the operator on the mapping.

Next, we need to make sure the data type is correct. The position value to which we're going to map this constant in the **SUBSTR** operator is defined as a **NUMBER** with no precision or scale specified (that is, both set to zero).

We are not going to bother specifying a precision or scale for the constants we're creating because we don't need the extra data integrity checks in the database and the **SUBSTR** `POSITION` attribute is defined that way. We'll leave the precision and scale set to zeros, which is the default.

Download at Boykma.Com

We need another constant value defined to indicate the length of the substring, so let's add another attribute on the **Output Attributes** tab of the **CONSTANT Editor** dialog box. We'll click on the **Add** button once more and change the name of this attribute to **LENGTH** to reflect its purpose. We'll leave the data type set to **NUMBER**, and the default precision and scale set to zero as we did for the `POSITION` attribute. We'll click on the **OK** button on the dialog box to close it.

We did not have a chance to specify the actual values of the two constant values we just entered because the **Editor** dialog box does not include that information. This is set via another property of the attributes, which is available to us in the **Attribute Properties** window on the left of the **Mapping Editor**. So we'll click on the first attribute in the **CONSTANT** operator, the **POSITION**, and look for the attribute property called **Expression**. It just so happens to be the very first attribute in the window, so let's click in the blank box to the right of **Expression**. In this field, we will enter the value to be set to this attribute. We'll enter **3** to indicate that we want the position of the substring to start at the third position of the source string. We can also use the **Expression Builder** by clicking on the button with three dots (**...**). But as this is a simple constant, we'll just enter **3** in the box.

Now we'll do the same thing for the `LENGTH` attribute and we'll enter a length of **4**, which is the length of the county ID portion of the `STORE_NUMBER`.

The next step is to connect our constants to the corresponding attributes of our **SUBSTR** operator. We'll drag a line from **POSITION** in the **CONSTANT** operator to the **POSITION** attribute of the **SUBSTR** operator, and from the **LENGTH** attribute to the **SUBSTRING_LENGTH** attribute.

Adding a TO_NUMBER transformation

The **SUBSTR** value is ready and we can use it to look up the county ID, but there's one more transformation we need to apply before we can use it to look up the county name. First, it needs to be converted into a number to match the data type of the ID field in the `COUNTIES_LOOKUP` table. To do this, we will use the **TO_NUMBER()** function. So let's drag a **Transformation Operator** onto our mapping to the right of the **SUBSTR** operator and select **TO_NUMBER** from the resulting pop up.

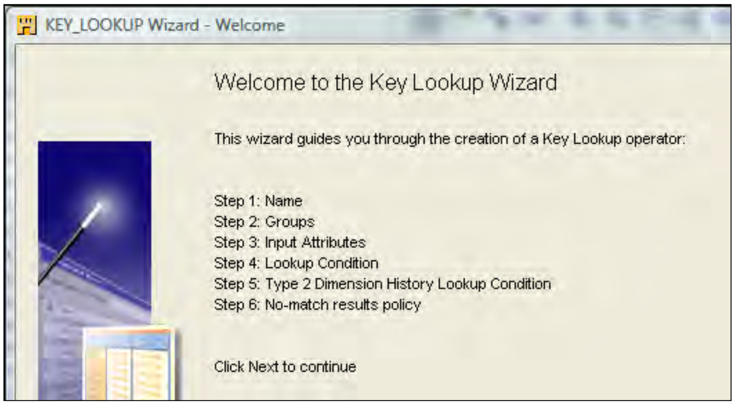
This operator needs three parameters, only one of which is absolutely necessary—the expression we wish to convert to a number. The other two parameters are optional and include a format string that we can use if we have a particular format of number we want (such as a decimal point in a certain place) and a parameter that allows us to set a certain national language format to default to if it's different from the language set in the database. We'll just map the input expression because our number is a straight integer format number. So let's drag a line from the **VALUE** attribute of **SUBSTR** to the **EXPR** input attribute of the **TO_NUMBER** operator.

Download at Boykma.Com

We are now ready to look up the number to find the county name. The final step we need to perform now is to actually add the Key Lookup operator that we'll use to do that, so let's continue with that task.

Adding a Key Lookup operator

After that little side trip to quickly create our lookup table and add a **SUBSTR** operator with a **TO_NUMBER** transformation to convert the result to a number, we can now add a **Key Lookup** operator to our mapping for looking up the county name. Let's drag a **Key Lookup** operator onto the mapping and drop it to the right of the **TO_NUMBER** operator. We can find the **Key Lookup** operator in the **Palette** window just as we did for the other operators we've added. After we drop it in the mapping, the **KEY_LOOKUP Wizard** is launched and it presents us with the **Welcome** screen. The wizard will guide us step-by-step through the process of defining our key lookup. It is composed of six steps, which we can see on the opening **Welcome** screen. A portion of this screen is shown here:



Welcome to the Key Lookup Wizard

This wizard guides you through the creation of a Key Lookup operator:

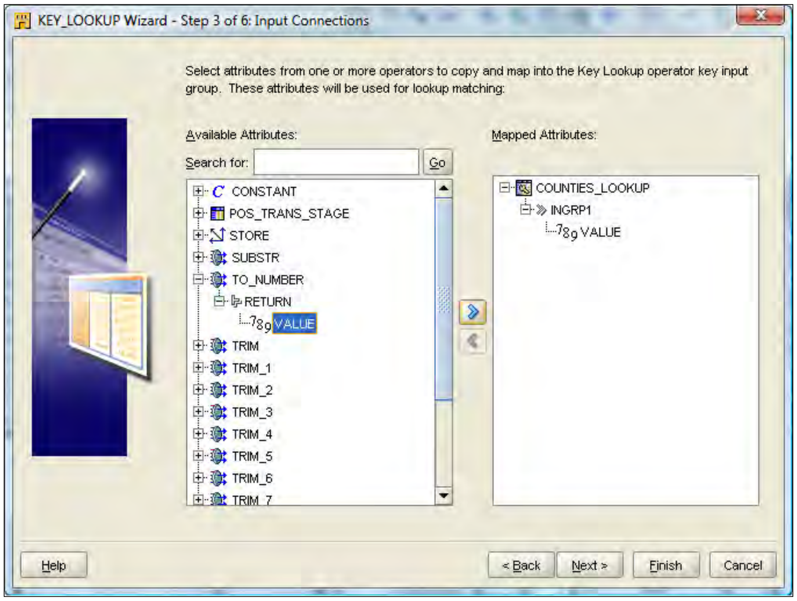
Step 1: Name
Step 2: Groups
Step 3: Input Attributes
Step 4: Lookup Condition
Step 5: Type 2 Dimension History Lookup Condition
Step 6: No-match results policy

Click Next to continue

1. After the welcome screen, the first step asks us for a name for this Key Lookup. It has a default name of **KEY_LOOKUP**. We'll change it to **COUNTIES_LOOKUP** and click on the **Next** button to proceed to the screen labeled **Step 2**.
2. This screen indicates that Key Lookup operators require one input and one output group, and here we have an opportunity to rename the groups if we desire. We'll leave them with their default names **INGRP1** and **OUTGRP1**, and click on **Next** to continue.

Download at Boykma.Com

3. The next screen labeled **Step 3** asks us to select one or more operators to map into the input group of the Key Lookup operator. This is where we will specify that the output (or return value) of the **TO_NUMBER** operator should be the value to use as input. We will choose the output attribute from that operator and it will create an input operator in the Key Lookup to match it. We'll look for the **TO_NUMBER** operator in the left window. We will scroll it down if it's not visible and expand it by clicking on the plus sign, and then we will expand the **RETURN** entry by clicking on the plus sign. We'll click on the **VALUE** attribute of the **TO_NUMBER** operator to select it and will click on the right arrow (**>**) to assign it to the **INGRP1** of the **COUNTIES_LOOKUP** Key Lookup operator as shown in the following screenshot:


KEY_LOOKUP Wizard - Step 3 of 6: Input Connections

Select attributes from one or more operators to copy and map into the Key Lookup operator key input group. These attributes will be used for lookup matching:

Available Attributes:

Search for:

Mapped Attributes:

COUNTIES_LOOKUP
INGRP1
VALUE

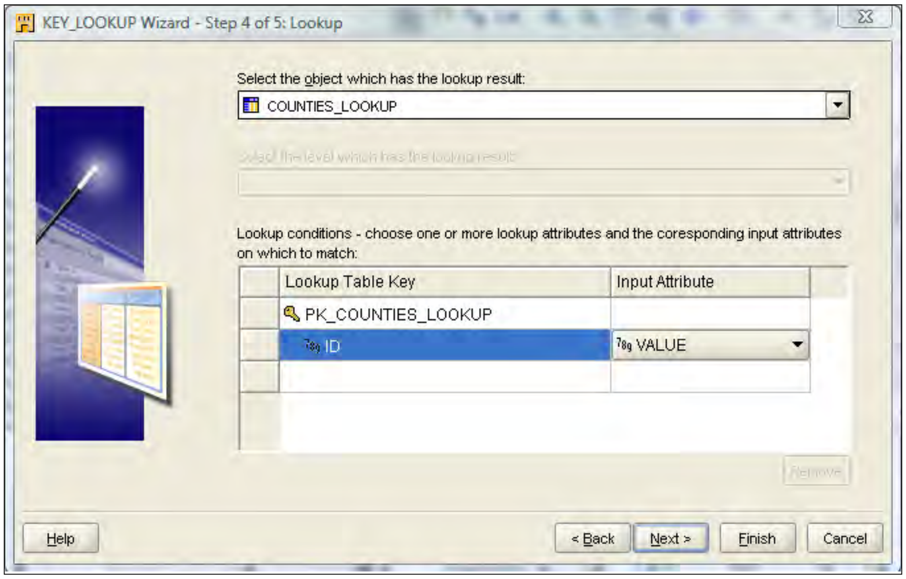
4. Now we've indicated that we want to use the output value from our **TO_NUMBER** operator to be the input value (or key) for the lookup operator. We'll click on **Next** and in the screen labeled **Step 4** we will specify in which object to look up the value.
5. The object can be a table, view, dimension, or cube object. In the screen labeled **Step 4**, we will select our new lookup table. So clicking on the drop-down menu at the top, we expand the **ACME_DWH** entry and see that it lists all the available tables, views, dimensions, and cubes in our project. We'll select the **COUNTIES_LOOKUP** table.

Download at Boykma.Com

6. We'll then click in the first row of the **Lookup Table Key** column. In the resulting drop-down menu that appears (which may take a moment or two to appear, so we'll be patient), we'll select the primary key we defined on the table. We could also have selected an individual column if we did not have a primary key on the table.
7. Having selected the primary key, we now have to specify an input attribute. We'll click in that box and see that it has added a row beneath with the **ID**, which shows as the column to use for the Lookup Table Key. Now we need to select the **Input Attribute** in the row in order to select the column from input that we want to use to match to this key column. This would be the **VALUE** attribute from our **TO_NUMBER** operator that we used in step 3, so we'll select that from the resulting drop-down menu.

It may seem redundant to make this selection here, but there could be more than one attribute used in a lookup. Therefore, we have to go through this step to indicate which input attribute matches with which lookup table key. In this case, we happen to have only a single attribute to use for the lookup.

Now our dialog box should look similar to the following:


KEY_LOOKUP Wizard - Step 4 of 5: Lookup

Select the object which has the lookup result: COUNTIES_LOOKUP

Lookup conditions - choose one or more lookup attributes and the coresponding input attributes on which to match:

Lookup Table Key	Input Attribute
PK_COUNTIES_LOOKUP	
ID	VALUE

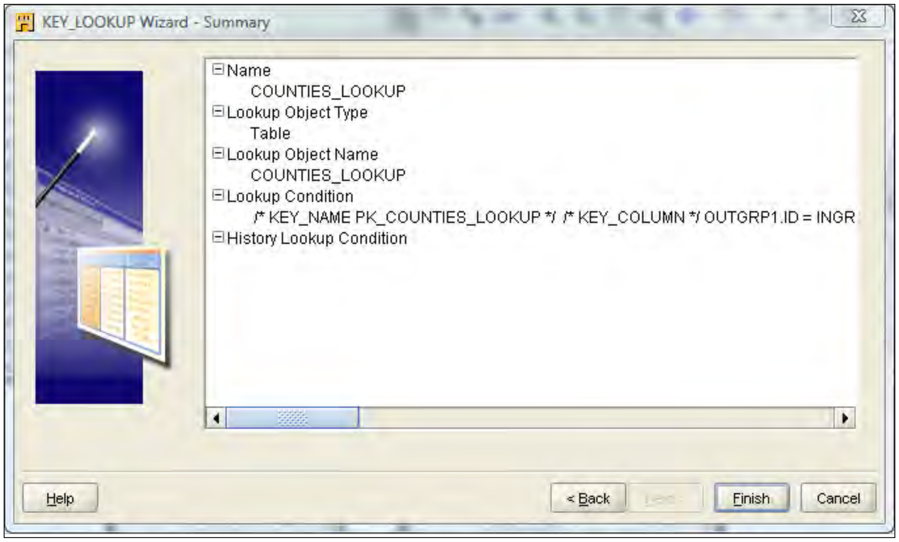
Download at Boykma.Com

8. We will click on the **Next** button to proceed to the final step where we will specify what to return if no record is found in the lookup table. Here we are only concerned with the `COUNTY_NAME` column as that is the value we need to map to the `SALES` cube. We'll specify a default value of `UNKNOWN` rather than just leave it `NULL`. So we'll click on **NULL** that currently appears as the default for the **COUNTY_NAME** value and type in **'UNKNOWN'** in the box.

> We have to make sure we include the single quotes around this string because it is a character string and the Oracle Database requires single quotes around character literals.

It also has an editor available to give us more power over the expression we might want to use to determine the value. But in our case, we only want a single string to be used, so we can just type it in.

9. We will click on the **Next** button to proceed to the **Summary** screen. It should look similar to the following screenshot:



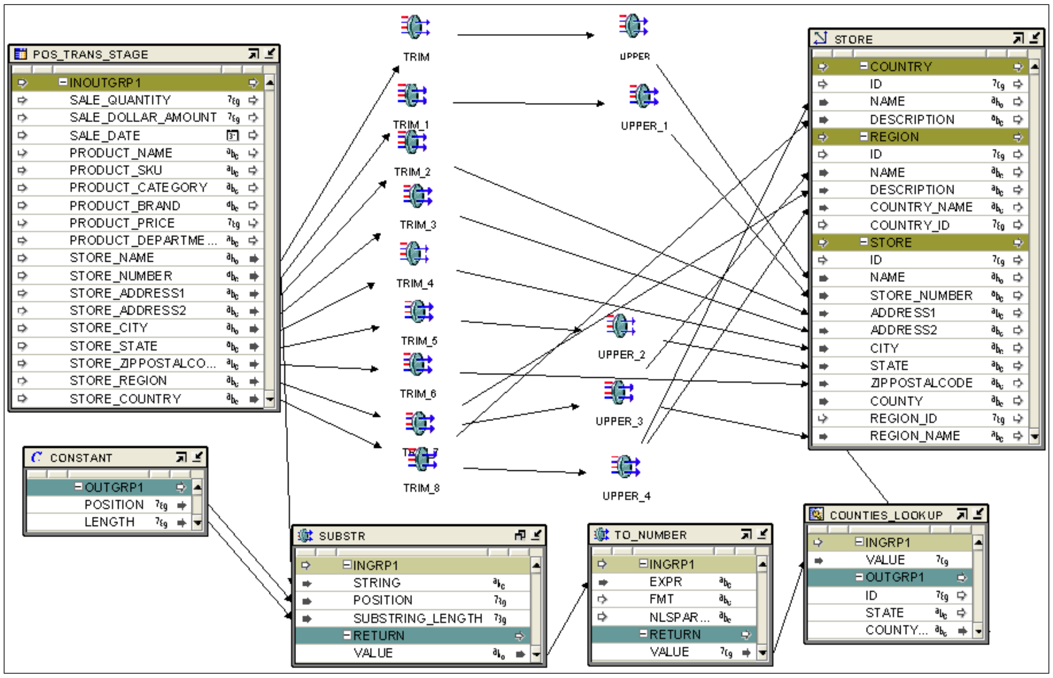
Name
COUNTIES_LOOKUP
Lookup Object Type
Table
Lookup Object Name
COUNTIES_LOOKUP
Lookup Condition
/* KEY_NAME PK_COUNTIES_LOOKUP */ /* KEY_COLUMN */ OUTGRP1.ID = INGR
History Lookup Condition

10. We will click on the **Finish** button and the wizard ends and drops a Key Lookup operator on our mapping with a connection line already drawn from the output attribute of the **TO_NUMBER** operator.

11. Now connect the **COUNTY_NAME** field from this Key Lookup operator to the **COUNTY** attribute in the **STORE** level of the **STORE** dimension and we are done with this mapping.

Download at Boykma.Com

Now we have a completed mapping that will populate our **STORE** dimension. Our final mapping should look similar to the following screenshot:



Of course, there are an almost infinite number of ways we could have organized our mapping. The mapping we just saw was somewhat compressed to better fit the available size for the image, so we won't focus on making the mapping look exactly like that. The important thing is that all the connections are made as they are shown in the mapping and not where each individual operator appears on the mapping.

Having completed our `STORE` mapping, we'll save our work with the *Ctrl+S* key combination. Now we need to move on to address the mapping for our `PRODUCT` dimension.

PRODUCT mapping

The mapping for the `PRODUCT` dimension will be similar to the `STORE` mapping, so we won't cover it in as much detail here. We'll open the Design Center if it's not already open and create a new mapping just as we did for the `STORE` mapping earlier and the `STAGE_MAP` mapping from the last chapter. We'll name this mapping `PRODUCT_MAP`.

Download at Boykma.Com

The source for the data will again be our staging table, POS_TRANS_STAGE, just as it was for the STORE mapping. Only the target will change as we're loading the PRODUCT dimension this time. We'll drag the **POS_TRANS_STAGE** table from the **Explorer** window and drop it on the left of the mapping, and drag the **PRODUCT** dimension from **ACME_DWH | Dimensions** and drop it to the right of the mapping. Not surprisingly, the data elements we'll now need from the staging table are the attributes that begin with PRODUCT. We created our PRODUCT dimension with four levels—DEPARTMENT, CATEGORY, BRAND, and ITEM—which we will need to populate. Let's start with the **ITEM** level and jump right to listing the attributes from the source to the target along with the issues we'll have to address with the data elements for this level:

- PRODUCT_NAME to NAME (in the ITEM level)—needs trimmed spaces and conversion to uppercase
- PRODUCT_NAME to DESCRIPTION (in the ITEM level)—needs trimmed spaces

We don't have a separate description field to map from the source. So for now we'll just map the name to it, but without converting to uppercase as we did for the STORE mapping. We'll do that for each of the other levels where description also appears.

- PRODUCT_SKU to SKU—needs trimmed spaces and conversion to uppercase
- PRODUCT_PRICE to LIST_PRICE—no transformation needed
- PRODUCT_BRAND to BRAND_NAME—needs trimmed spaces and conversion to uppercase

We'll add the needed transformations to accomplish the changes as indicated in the list we just saw, and then move on to the BRAND level. For the BRAND level, we need to map the NAME, DESCRIPTION, and CATEGORY_NAME as follows:

- PRODUCT_BRAND to NAME (in the BRAND level)—needs trimmed spaces and conversion to uppercase
- PRODUCT_BRAND to DESCRIPTION (in the BRAND level)—needs trimmed spaces
- PRODUCT_CATEGORY to CATEGORY_NAME—needs trimmed spaces and conversion to uppercase

Download at Boykma.Com

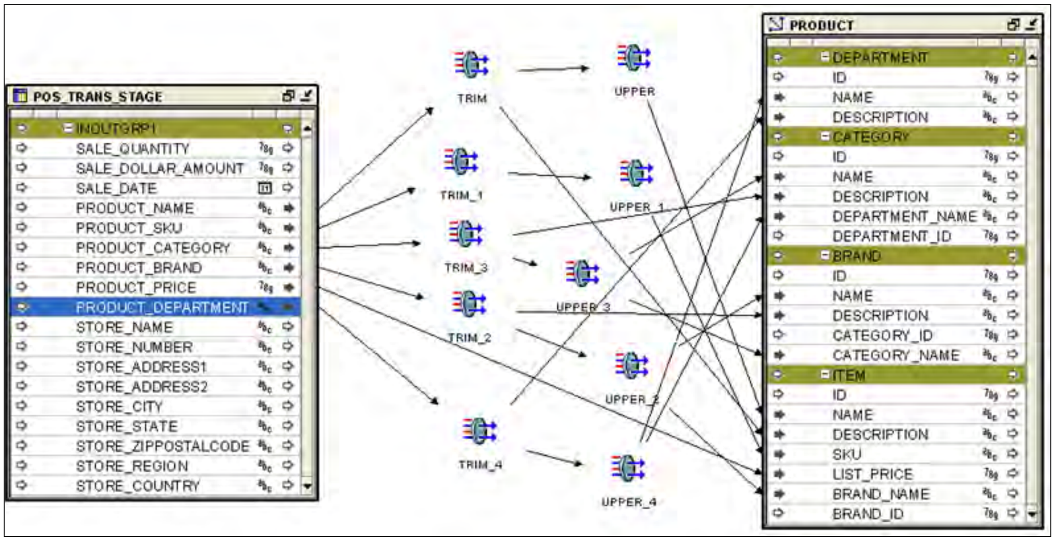
When we have added these transformations and made these connections to the BRAND level, we'll move on to the CATEGORY level. It will be mapped in a similar manner to BRAND, but using the PRODUCT_CATEGORY attribute as input.

- PRODUCT_CATEGORY to NAME (in the CATEGORY level)—needs trimmed spaces and conversion to uppercase
- PRODUCT_CATEGORY to DESCRIPTION (in the CATEGORY group)—needs trimmed spaces
- PRODUCT_DEPARTMENT to DEPARTMENT_NAME—needs trimmed spaces and conversion to uppercase

Finally, we'll map the DEPARTMENT level, which has just two attributes we need to be concerned about—the NAME and the DESCRIPTION.

- PRODUCT_DEPARTMENT to NAME (in the DEPARTMENT level)—needs trimmed spaces and conversion to uppercase
- PRODUCT_DEPARTMENT to DESCRIPTION (in the DEPARTMENT level)—needs trimmed spaces

When we have completed these additional connections and transformations, we will have completed the mapping for the PRODUCT dimension. It should look similar to the following screenshot, which shows all the transformations and connections in place that were described earlier:



Download at Boykma.Com

We have conserved on the usage of Transformation Operators by making multiple connections from some of them as we did for the STORE_MAP previously. For instance, the topmost **TRIM** has a connection to the **UPPER** transformation to convert its output to uppercase before connecting to the **NAME** attribute of the **ITEM** group. But it also connects directly to the **DESCRIPTION** attribute of the **ITEM** group. It was not necessary to have the description in all uppercase, so the output of the **TRIM** was used. We could have just as easily dragged another **TRIM** as well as another line from the **PRODUCT_NAME** in **POS_TRANS_STAGE** to the mapping, but we would have ended up with more clutter than necessary. The functioning of the mapping would have been the same in either case.

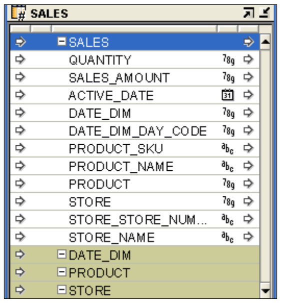
We have completed the mapping for our PRODUCT dimension, and that completes all the mappings for our dimensions we will need to do. There is a third dimension we'll be using, the DATE_DIM mapping, but that mapping was created automatically for us. Let's save our work with the *Ctrl+S* key combination, or with **Design | Save All** from the **Design Center** main menu, or with **Mapping | Save All** from the **Mapping Editor** main menu. Moving on, we'll now create a mapping to populate our cube and that will be all the mappings we'll need for our data warehouse.

SALES cube mapping

Turning our attention to the cube, we have one more mapping to create. It will be created in the same way as we created the previous maps, but let's call this one SALES_MAP. In this mapping, we will need to draw data from the POS_TRANS_STAGE table as input as we did for other two dimension maps, and we will have the SALES cube as the output target to load our data. Let's drag each of these onto our mapping using **Table Operator** for the POS_TRANS_STAGE table and **Cube Operator** for the SALES cube.

Download at Boykma.Com

The POS_TRANS_STAGE table is very familiar to us as we have used it for the two dimensions, but the SALES cube is new. It looks slightly different than the dimensions we worked with earlier in this chapter, so let's take a moment to go over it in a little more detail. When dropped onto our mapping and expanded completely, it should look similar to the following:



The top group with visible attributes is the main group for the cube and contains the data elements to which we'll need to map. The other groups represent the dimensions that are linked to our cube. We mapped data to these dimensions (except for the DATE_DIM dimension) earlier. So there is no need to map any data to these groups in the cube now and, indeed, it doesn't even provide a way to do that here. The data we map for the dimensions will be to attributes in the main cube group, which will indicate to the cube which record is applicable from each of the dimensions.

Dimension attributes in the cube

Each dimension is represented in the cube attributes by a surrogate identifier, which is the primary key for the dimension, and the business identifier(s) defined for the dimension. This is where we will see the real usefulness of the business identifiers that we specified when we designed our dimensions in *Chapter 4*. They identify the dimension record for this cube record and the surrogate identifier will be used as the foreign key to actually link to the appropriate dimension record in the database. Let's take a look at these attributes for the dimensions.

For the PRODUCT dimension, we have seen three attributes earlier that are product related—PRODUCT_NAME, PRODUCT_SKU, and PRODUCT. If we were to open our PRODUCT dimension in the Data Object Editor to view its attributes, we wouldn't see any attributes with exactly these names. The Warehouse Builder has provided us with attributes that represent the corresponding attributes from the dimension, but with a slightly different naming scheme. The PRODUCT_NAME and PRODUCT_SKU attributes correspond to the NAME and SKU attributes from the dimension that are the business identifiers we defined. The PRODUCT attribute corresponds to the ID attribute that was created automatically for us as the surrogate identifier for the dimension.

Download at Boykma.Com

The Warehouse Builder uses a convention for all the dimension attributes in the cube. It prefixes the name of the dimension onto the name of the attribute for the business identifiers and uses the name of the dimension as the surrogate identifier. The usage of the same name in more than one dimension is always the case with the ID attribute and frequently the case with other attributes we define. Prefixing the attribute with the name of the dimension will make it unique when included in the cube, so we'll be able to tell to which dimension it matches.

Another reason is that the underlying implementation for the cube in the database uses relational tables when ROLAP is selected for the storage option, and the database will not allow the same name to be used for more than one column in a table.

Renaming dimension attributes

The names for dimension attributes in cubes that OWB comes up with can be changed if we desire. We would just have to right-click on the name in the cube operator in the mapping and select **Open Details...** from the pop-up menu. On the **Input/Output** tab, we would just click on the name and type a new name. Of course, we would have to make sure we didn't choose something already taken. But if we do, the Warehouse builder will definitely let us know immediately by popping up an error dialog box, and will change the name right back to what it was and give us another chance.

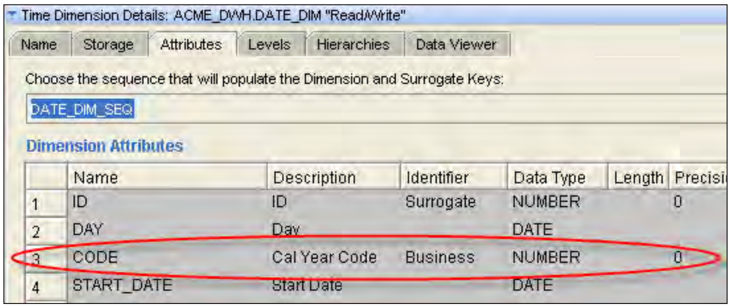
For the `STORE` dimension, we identified `NAME` and `STORE_NUMBER` as the business identifiers. Looking at the list of attributes for the `STORE` dimension that OWB created for us, we see `STORE_NAME` and `STORE_STORE_NUMBER`. Remember about the prefixing of the dimension name. In most cases that will work out OK, but in some cases where we might have included the dimension name as a part of our attribute name in the dimension (as we did for the `STORE_NUMBER`), we will end up with the dimension name repeated twice. It doesn't hurt anything to leave it as is; but if we want to change it, we can simply use our tip above about editing the name. Let's go ahead and make that change. So now we have `STORE_NUMBER` instead of `STORE_STORE_NUMBER`. We also have a third attribute, which we recall is for the surrogate identifier. It is named for the dimension, which is `STORE` in this case.

Another clue that can help us figure out what attribute is what is to look at the type that was defined for the attribute. The type icons just to the right of the name in the Cube Operator show us that the `STORE` and `PRODUCT` attributes are numeric. From our dimension design we know that the ID surrogate identifier was also defined as numeric.

Download at Boykma.Com

The third dimension we have defined for our cube is the `DATE_DIM` dimension. In this chapter, we have been focusing mainly on the `PRODUCT` and `STORE` dimensions because we had to define them ourselves, but we still have to work with the `DATE_DIM` dimension because it's also a part of the cube. We can see two attributes defined for it, both of them numeric fields. We can see that one field is named for the dimension—`DATE_DIM`. So we can surmise that it is for the surrogate identifier for the dimension. But the other attribute—`DATE_DIM_DAY_CODE`—is not so obvious. One issue is that we didn't create that dimension manually; it was created for us by the **Time Dimension Wizard**. So we are not as familiar with its attributes as we might be with the other two dimensions we defined ourselves. However, the Warehouse Builder is following the same naming convention for the `DATE_DIM` dimension as for the other two, so this attribute must be the business identifier for the dimension.

Let's verify the business identifier for the `DATE_DIM` dimension by opening it in the Data Object Editor. To open it in the Data Object Editor, navigate in the Design Center to **ACME_DWH | Dimensions |DATE_DIM** and double-click to open it. Looking at the **Attributes** tab, we see that there is a business identifier called **CODE** as shown in the following screenshot:



Time Dimension Details: ACME_DWH.DATE_DIM "Read/Write"

Choose the sequence that will populate the Dimension and Surrogate Keys:

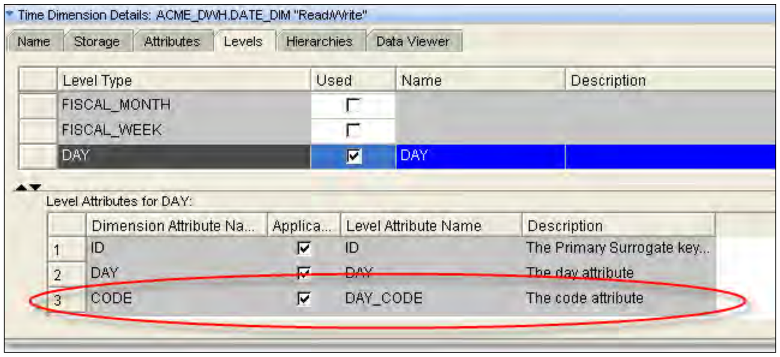
DATE_DIM_SEQ

Dimension Attributes

	Name	Description	Identifier	Data Type	Length	Precisi
1	ID	ID	Surrogate	NUMBER		0
2	DAY	Day		DATE		
3	CODE	Cal Year Code	Business	NUMBER		0
4	START_DATE	Start Date		DATE		

If OWB is following the same naming convention here, we would think that this attribute should be named `DATE_DIM_CODE` and not `DATE_DIM_DAY_CODE`, but where did it get the "`DAY`" from in the name? Looking a little further, let's look at the **Levels** tab where we can inspect each level of the dimension and see what attributes were identified for each level. We can scroll down and click on the **DAY** level in the window in the top portion of the **Levels** tab and we are presented with the **Level Attributes** for the **DAY** level in the scrolled window at the bottom of the tab. There we can see the **CODE** attribute once again as shown in the following screenshot:

Download at Boykma.Com



Time Dimension Details: ACME_DWH.DATE_DIM "Read/Write"

Name | Storage | Attributes | Levels | Hierarchies | Data Viewer

Level Type	Used	Name	Description
FISCAL_MONTH	☐		
FISCAL_WEEK	☐		
DAY	✓	DAY	

Level Attributes for DAY:

	Dimension Attribute Na...	Applica...	Level Attribute Name	Description
1	ID	✓	ID	The Primary Surrogate key...
2	DAY	✓	DAY	The day attribute
3	CODE	✓	DAY_CODE	The code attribute

This time, however, we see something different than what we've seen with our other two dimensions. The **Level Attribute Name** for the `CODE` attribute on the `DAY` level is `DAY_CODE`. The **Time Dimension Wizard** has prefixed the level name onto the attribute name. If we were to look at the other levels, we'd see the `CODE` named the same way with the level name prefixed to it—`CAL_MONTH_CODE`, `CAL_QUARTER_CODE`, and `CAL_YEAR_CODE` for the `CALENDAR_MONTH`, `CALENDAR_QUARTER`, and `CALENDAR_YEAR` levels. So it has taken the lowest level and used the attribute name for it, and prefixed the dimension name to come up with the `DATE_DIM_DAY_CODE` name. It used this level because when we defined the cube, the `DAY` level was the level specified for referencing the `DATE_DIM` dimension. This can be verified in the Data Object Editor when viewing the **SALES** cube in the **Dimensions** tab.

For future projects that are more complicated, we will definitely want to think about implementing a similar naming convention for any dimensions we define. It's possible to have our cubes reference a dimension at a different level, and not just at the bottom level. It would make it much easier with one glance at our cube to tell at which level the cube is referencing the dimension. In our case with the ACME Toys and Gizmos company data warehouse, we are sticking with the default level and so it's not as big an issue.

Measures and other attributes in the cube

Two of the remaining attributes we can see in the `SALES` cube are the measures we defined for our cube—the quantity of the items sold and the dollar amount of the sale. We specified these names explicitly in the **Cube Wizard** when we defined our cube in *Chapter 4*, so they appear here as we named them.

Download at Boykma.Com

We can see one final attribute that we haven't accounted for yet, and that is called ACTIVE_DATE. It's created automatically for us and is designed to support Type 2 **Slowly Changing Dimensions** (**SCD**). It is used as the time to determine the active record in a Type 2 SCD. This is a more advanced topic and there is a more thorough explanation of this field in the *Oracle Warehouse Builder Users Guide* at http://download.oracle.com/docs/cd/B28359_01/owb.111/b31278/toc.htm. If you would like get more information about this field, read Chapter 17 on Source and Target operators of this document where the Cube operator is discussed. For our purpose, we don't need to do anything with it as we have no Type 2 SCDs. If we don't map anything to this field, the Warehouse Builder will simply populate it using the SYSDATE Oracle function, which sets it to the current system date/time.

Mapping values to cube attributes

Now that we've taken a look at the attributes in our cube, we need to turn our attention to getting values mapped to them. We'll begin by mapping values for the measures because they are the simplest, and then proceed to map values for the two dimensions we created mappings for earlier, the PRODUCT and STORE. We saved the DATE_DIM dimension for last because this will introduce us to another operator type that we haven't seen yet, and this operator will be needed to derive the code for the DATE_DIM dimension.

Mapping measures' values to a cube

The measures that get mapped to a cube are most often numbers, so we don't have to be concerned with the TRIM and UPPER operators for them. Sometimes we may need to do calculations on values from input before storing them in output, but in our case now we just want to map the values from the input as they are. The SALE_QUANTITY from our POS_TRANS_STAGE table is going to provide the value for the QUANTITY in the SALES cube, and the SALE_DOLLAR_AMOUNT is going to supply the value for the SALES_AMOUNT attribute.

We'll drag a line directly from SALE_QUANTITY to QUANTITY, and another line directly from SALE_DOLLAR_AMOUNT to SALES_AMOUNT. This completes the measures. We saw earlier that the ACTIVE_DATE attribute didn't need anything mapped to it, so we'll move right to the dimensions.

Download at Boykma.Com

Mapping PRODUCT and STORE dimension values to the cube

When mapping values for a dimension in a cube, we only need to concern ourselves with mapping the business identifiers. The Warehouse Builder will take care of the lookup to determine the identifier to fill in for the key value (surrogate identifier), so we don't have to worry about it. We get this for free by using OWB's Cube and Dimension Operators where we'd have to do a manual lookup if we were to use regular Table Operators.

So for the `PRODUCT` dimension-related attribute values, we have the `PRODUCT_NAME` and `PRODUCT_SKU` to which we need to map values to. We can leave the `PRODUCT` attribute alone as it will be populated automatically behind the scenes based on a lookup using the values we provide for name and SKU.

> The `PRODUCT_NAME` and `PRODUCT_SKU` attributes were chosen as business identifiers because they uniquely identify a single product. No two products in the ACME Toys and Gizmos company inventory are assigned the same name and SKU. This is why they can be used here to look up the dimension record key for us.

Looking back at our source table now, which is the `POS_TRANS_STAGE` mapping table, we need to find the name and SKU attributes to map to the corresponding attributes in the cube.

> Here it's very important to make sure whether there were any transformations applied to the values when mapping them to the dimension in the dimension mapping. If there were transformations, the same transformations need to be applied here to ensure a match will be made.
>
> If we had a name in mixed case and stored it in the dimension using the `UPPER` transformation, but used the value without applying the `UPPER` to it for a lookup, we would not get a match.

Earlier in this chapter where we mapped these fields in the dimension mappings, we decided we would apply a `TRIM` and `UPPER` to them before storing. So this is what we'll need here also. Our Mapping Editor must be already open from when we dragged in the source and target operators. So let's now drag a Transformation Operator to our mapping and make it a **TRIM**, and drag a second Transformation Operator to our mapping and make it an **UPPER**. We'll connect the **PRODUCT_NAME** from our source to the input of the **TRIM**, the output of the **TRIM** to the input of the **UPPER**, and the output of the **UPPER** to the **PRODUCT_NAME** attribute in the cube. The `PRODUCT_SKU` attribute needs to be mapped in the same way to the `PRODUCT_SKU` attribute of the `SALES` cube.

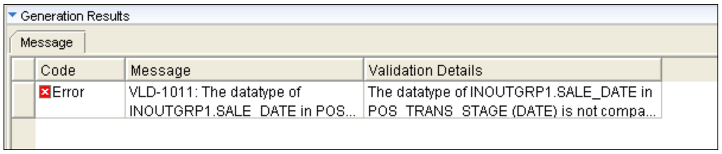
Download at Boykma.Com

We'll do the same steps for the STORE_NUMBER and STORE_NAME fields from input to the same named attributes in the cube. This will take care of these two dimensions. The PRODUCT and STORE attributes in the cube will be automatically populated with the key value for the corresponding dimension record. Finally, we'll look at the DATE_DIM dimension values for mapping.

Mapping DATE_DIM values to the cube

We saw earlier that the DATE_DIM_DAY_CODE was the business identifier value for the DATE_DIM dimension, so we need to map a value to it. Looking at our POS_TRANS_STAGE mapping table for input, the only date-related value we have is the SALE_DATE field. However, the DATE_DIM_DAY_CODE field is not defined as a DATE field; it's defined as a NUMBER field. We can't just drag a date field to a number field. Well, we can (the Mapping Editor will let us), but when we **validate** the mapping we'll get an error. We will discuss the concept of validation in greater detail in the next chapter. It is the process the Warehouse Builder uses to check a mapping for various error conditions. Date fields cannot be directly converted into numbers in the Oracle Database, and so that would generate an error.

To illustrate, let's try to map the date directly to the number field and see what kind of error we get. We'll drag a line from the SALE_DATE attribute of POS_TRANS_STAGE to the DATE_DIM_DAY_CODE attribute of the SALES cube. Then we'll validate the mapping by selecting the **Validate** menu entry under the **Mapping** main menu, or by pressing the *F4* key. The result, as it initially appears in the **Generation Results** window at the bottom of the **Mapping Editor**, is as follows:

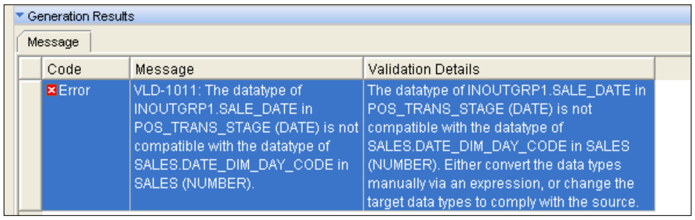


Generation Results

Message

Code	Message	Validation Details
Error	VLD-1011: The datatype of INOUTGRP1.SALE_DATE in POS...	The datatype of INOUTGRP1.SALE_DATE in POS_TRANS_STAGE (DATE) is not compa...

To see the entire message, expand the display by double-clicking on it. An example of this error is shown in the following screenshot:



Generation Results

Message

Code	Message	Validation Details
Error	VLD-1011: The datatype of INOUTGRP1.SALE_DATE in POS_TRANS_STAGE (DATE) is not compatible with the datatype of SALES.DATE_DIM_DAY_CODE in SALES (NUMBER).	The datatype of INOUTGRP1.SALE_DATE in POS_TRANS_STAGE (DATE) is not compatible with the datatype of SALES.DATE_DIM_DAY_CODE in SALES (NUMBER). Either convert the data types manually via an expression, or change the target data types to comply with the source.

Download at Boykma.Com

As it is clear that we will not be able to connect the `SALE_DATE` directly to the cube, we must convert it somehow. Let's remove the connection we just made and then take a look at exactly what we're going to have to do instead. Click on the line just drawn and press the *Delete* key.

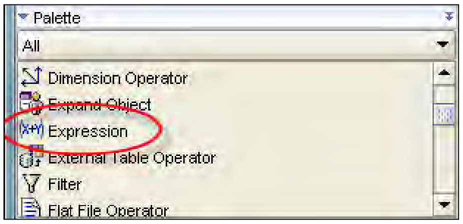
We have to turn the date field into a character representation of the date, that is, all numbers and that character value can then be converted to a number. Now the question is: What should be the format of that character representation of a date? In *Chapter 4* we discussed this issue and found the answer in the *Oracle Warehouse Builder Users Guide 11g Release 1*, Chapter 14 on Defining Dimensional Objects. In this manual, there is a *Using a Time Dimension in a Cube Mapping* section, which discusses this topic in good detail (`http://download.oracle.com/docs/cd/B28359_01/owb.111/b31278/ref_dim_objects.htm#BCGHJHIB`). The main point we take away from this section is that we need to apply the following expression to our date field to turn it into the number that is expected for that `DATE_DIM_DAY_CODE` field:

```
TO_NUMBER(TO_CHAR(input,'YYYYMMDD'))
```

The `input` in the expression we just saw will be the `SALE_DATE` value, and the `TO_CHAR()` function will convert that date into a character string of the format `'YYYYMMDD'` where `YYYY` is a four-digit year, `MM` is a two-digit month, and `DD` is a two-digit day. This provides us with the date expressed as all numbers. The `TO_NUMBER()` function then takes that character string of numbers and converts it into an actual number that we can store in the `DATE_DIM_DAY_CODE` field.

Mapping an Expression operator

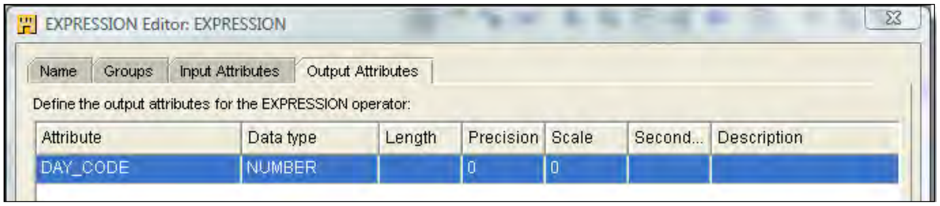
Let's now turn our attention back to the Mapping Editor and map the `SALE_DATE` to the cube using this expression. To include an expression like the above in our mapping, the Mapping Editor provides an **Expression** operator in the **Palette** Window as shown in the following screenshot:



Download at Boykma.Com

The steps to perform for the required expression are:

1. Drag the **Expression** operator onto our mapping, and drop it between our source and target operators. Initially, it doesn't display any defined attributes, but it does have two groups defined—an input group, `INGRP1`—and an output group, `OUTGRP1`. Expression operators are for defining any type of valid Oracle PL/SQL expression and, therefore, provide us tremendous flexibility in defining our own transformations, which we will do now.
2. Let's drag the **SALE_DATE** from the **POS_TRANS_STAGE** mapping table to the **INGRP1** of the **EXPRESSION** operator we just dropped into our mapping. This will now make the `SALE_DATE` available to us to use in the expression.
3. First, we need to create an output attribute that will hold the results of our expression. It will be used to map to the cube. Right-click on **OUTGRP2** and select **Open Details...** from the pop-up menu.
4. This will display an **Editor** window for the expression, so we'll click on the **Output Attributes** tab because we need to add an attribute as output. It is initially blank as no attributes are defined, so we click on the **Add** button to add an attribute. It will add an attribute with a default name of **OUTPUT1** and a default data type of **NUMBER**. The default data type in this case happens to be exactly what we need. This value will be mapped to the **DATE_DIM_DAY_CODE** attribute of the **SALES** cube, and this one is also defined as a number.
5. We could leave the name as the default of **OUTPUT1**, but that is so non-descriptive! Let's change it so that it indicates better what value it will be creating for us. We'll name it **DAY_CODE** by clicking on the name and just typing in the new name in the **EXPRESSION Editor**. When we're done, it should look similar to the following screenshot :

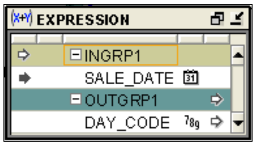


Define the output attributes for the EXPRESSION operator:

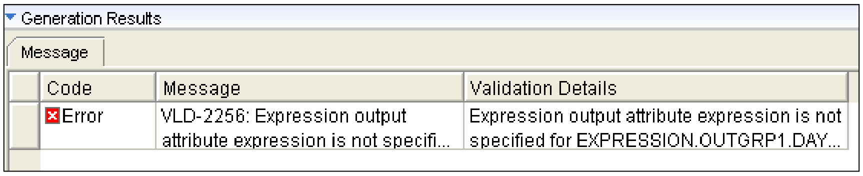
Attribute	Data type	Length	Precision	Scale	Second...	Description
DAY_CODE	NUMBER		0	0		

6. We'll click on the **OK** button to close the **Editor** window and now our **EXPRESSION** will look similar to the following when fully expanded:

Download at Boykma.Com

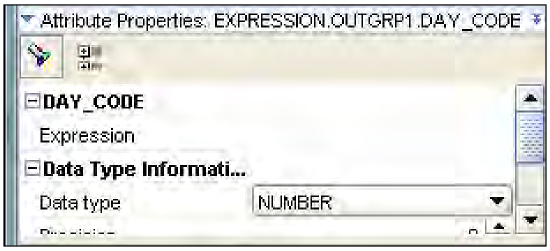


7. We can now drag a line from the **DAY_CODE** of our **EXPRESSION** to the **DATE_DIM_DAY_CODE** attribute of our **SALES** cube. However, we still might have a line connecting the **SALE_DATE** directly to the **DATE_DIM_DAY_CODE** attribute from our little investigation earlier about checking out the validation error. If so, we can just click on that line and press the *Delete* key, or right-click and select **Delete** to remove it. At this point, we are not quite done with mapping the expression because if we try to validate now, we'll get the following error:



Code	Message	Validation Details
Error	VLD-2256: Expression output attribute expression is not specifi...	Expression output attribute expression is not specified for EXPRESSION.OUTGRP1.DAY...

8. We have not yet told the Expression operator what expression it needs to use to produce the desired output. This is the expression we got out of the *User's Guide* that we saw earlier. In Expression operators, the expression to use is stored as a property of the output attribute that we've defined. So let's click on the **DAY_CODE** attribute in the **EXPRESSION** operator and turn our attention to the property window of the **Mapping Editor**. It is now labeled as **Attribute Properties**, which is shown in the following screenshot:



Data type: NUMBER

Download at Boykma.Com

9. The very first property shown is **Expression** where we will enter the expression to use. We'll click in the blank area to the right of the **Expression** label in the **Attribute Properties** window and it changes to allow us to enter text into that block. We could just enter it directly. For simple expressions this works well, but for most expressions we'll want to make use of the powerful **Expression Builder** tool that the Warehouse Builder provides to construct expressions. We can launch the **Expression Builder** by clicking on the button with three dots to the right of the **Expression** input field that appeared when we clicked on it. The **Expression Builder** dialog box appears and we can create our expression. We've seen this before in the previous chapter when we were specifying the join condition to use for the Joiner operator in our `STAGE_MAP`. It is basically the same dialog box, as join conditions are also simply expressions. We'll type in the following into the expression window labeled **Expression for DAY_CODE** to begin our expression:

   ```
   TO_NUMBER(TO_CHAR(
   ```

10. The next item to enter is the name of the input to use and for that we want to specify the `SALE_DATE` input attribute. This is the real benefit of using the Expression Builder. We can now just double-click on the **SALE_DATE** attribute under the **INGRP1** heading in the left pane to enter it into our expression at the point where the cursor is currently located, which should be right after the last open parenthesis we entered. We don't have to worry about using the correct syntax to specify it, as it gets entered for us.

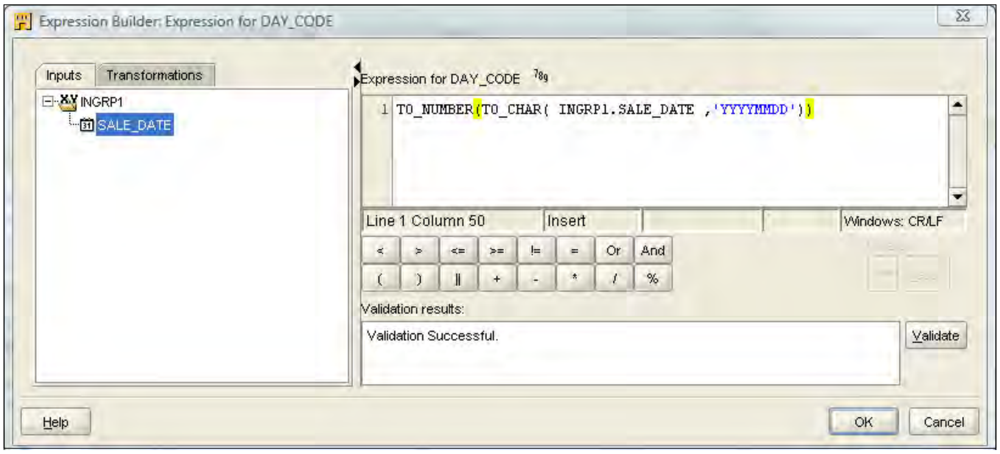
> We can notice a feature of the Expression Builder that when the cursor is just to the right of that last parenthesis, it turns red. This is the bracket matching feature of the Expression Builder. It helps us make sure we have corresponding closing parentheses for any open parentheses. When it finds a matching closing parenthesis, it will highlight them both in yellow.

11. We'll now finish entering our expression by typing in the following text to close out the expression:

   ```
   ,'YYYYMMDD'))
   ```

Download at Boykma.Com

12. We'll click on the **Validate** button to make sure our expression is correctly entered and our **Expression Builder** window will now look like the following screenshot:

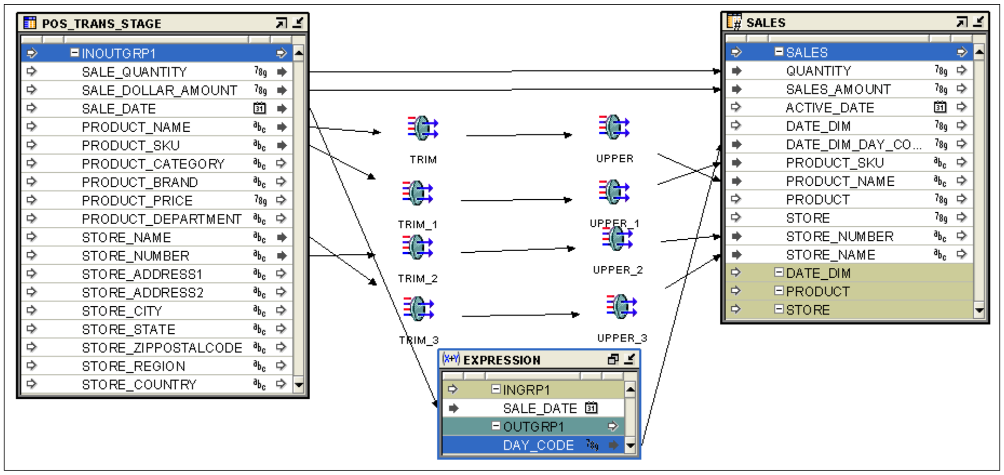


```
TO_NUMBER(TO_CHAR( INGRP1.SALE_DATE ,'YYYYMMDD'))
```

Validation Successful.

13. We'll click on the **OK** button to close the **Expression Builder** and our mapping now validates successfully.

With that we have completed mapping all the attributes that are needed for our cube. Our mapping should now look similar to the following with all our transformation operators collapsed into their icon view and the **EXPRESSION** operator left open:



Download at Boykma.Com

Our dimension mappings were completed earlier in the chapter, so we now have all the mappings completed that we'll need to populate our data warehouse. This means we are now ready to deploy them to the database and try them out. This will be the main topic of the next chapter.

Features and benefits of OWB

Before we move on to the next chapter, let's take a moment to recapitulate some of the features that the Oracle Warehouse Builder provides to us to make our job easier. This is why we made the choices we did for our design and implementation. By providing us with the option to implement our cubes and dimensions either relationally with ROLAP or fully multi-dimensionally with MOLAP, OWB allows us to design one way in OWB and implement either way in the database with a simple change of a storage option.

- By providing us the **ROLAP option**, the Oracle Warehouse Builder opens to us the design features of cubes and dimensions even though we'll be implementing them relationally with tables in our database. Choosing that option rather than just implementing tables directly saves us from having to worry about dimension keys, sequences to populate them, and providing lookups of dimension record keys to fill in for our cube. When loading a dimension, all we have to do is map data to it and it handles constructing the levels and assigning keys automatically. When mapping to the cube, all we have to do is specify business identifier attributes in our dimensions and map values to them in the cube. The rest is all handled for us.

  The underlying tables and sequences are all built automatically for us, so we need not be concerned with building any tables or sequences.
- The **Expression Builder** provides us a powerful tool to use for interactively building expressions.
- Support for **Slowly Changing Dimensions** is built in with the Enterprise ETL option. Although we didn't make use of that feature in this introductory book, the support is there if we need it and have paid for it when we build more advanced data warehouses and want to implement SCDs.

These are just some of the benefits the Warehouse Builder provides to us and were the basis for many of our decisions throughout the book regarding how we chose to implement our data warehouse using OWB.

Download at Boykma.Com

Summary

Now we are real close to actually having a data warehouse built in our database. We've completed all the mappings that we will need for the Warehouse Builder to create the code that will run to actually pull data from our source system, load our staging table, and then load our target data warehouse structure from our staging table.

We've seen how to use Transformation Operators to apply functions to our data to change it before loading. We've also seen the Expression operator for entering custom expressions. We've taken a look at how to connect them together with source and target operators to complete a mapping and successfully validate them.

Now we're ready to deploy these mappings to the database and execute them to actually load our data, which we will do in the next chapter.

Download at Boykma.Com

Download at Boykma.Com

8 Validating, Generating, Deploying, and Executing Objects

We have reached the last step in the process of building our data warehouse. We've done a large amount of work so far, which includes designing target schemas, creating objects in the Warehouse Builder, and creating mappings to load data into our target. However, we have yet to actually create a single real database object. What we have is the complete design stored in the Warehouse Builder. Let's think of ourselves as architects of new houses; only instead of designing a house, we're designing a data warehouse. Before any house can be physically built, someone has to design it and create a model of the house that the builders will then use for construction. That's what we've been doing up until this point, that is, creating the model of our data warehouse.

Now we are at the point where the model is complete and we're ready to actually build the data warehouse in an actual database, and load data into it. So, we get to be the builders also and not just the architects, and that is what this chapter is all about. The process of building the data warehouse from our model in the Warehouse Builder involves the following four steps:

- **Validating**: Checking objects and mappings for errors
- **Generating**: Creating the code that will be executed to create the objects and run the mappings
- **Deploying**: Creating the physical objects in the database from the logical objects we designed in the Warehouse Builder
- **Executing**: Executing the logic that is found in the deployed mappings for mappings and transformations

Download at Boykma.Com

The first three steps—validating, generating, and deploying—generally go together as all objects and mappings are deployed. A deployment will automatically do a validation and generation first before deploying. The fourth step—execution—is an independent process that is performed on those objects that contain ETL logic and mappings after they've been deployed. It doesn't happen for everything that we design in the Design Center. The Design Center has menu entries that will allow us to validate, generate, and deploy objects, but not execute them. We will be introduced to a new interface called the **Control Center Manager**, which is the tool for controlling the deployment of objects and execution of mappings.

Throughout the remaining chapter, the word "objects" will generally be used to refer to any type of object or mapping that can be built in the Warehouse Builder.

We will discuss each of the four processes separately in more detail in this chapter although we'll frequently find ourselves doing just deployments and executions, as the deployment process includes a validation and generation. We need to understand each of these processes, so let's get started by talking more about the validation of objects and mappings.

Validating

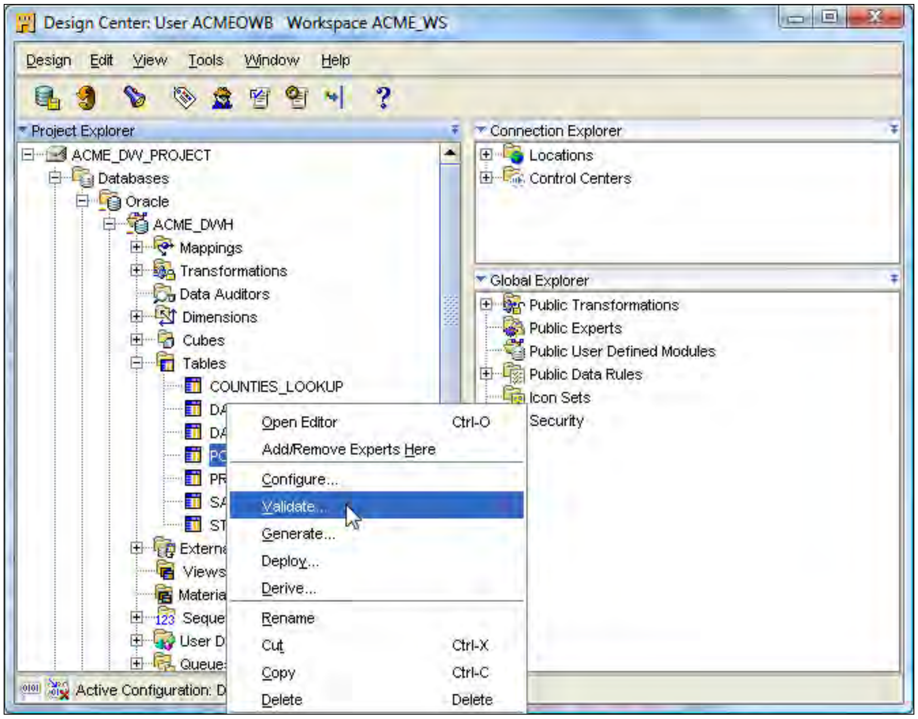
We briefly touched upon the topic of validation of mappings in the last chapter when we were working on our `SALES` cube mapping and talked about the error we would get if we tried to map a date attribute to a number attribute. Error checking is what validation is for. The process of validation is all about making sure the objects and mappings we've defined in the Warehouse Builder have no obvious errors in design.

Let's recap how we go about performing a validation on an object we've created in the Warehouse Builder. There are a number of places we can perform a validation. One of them is the main Design Center.

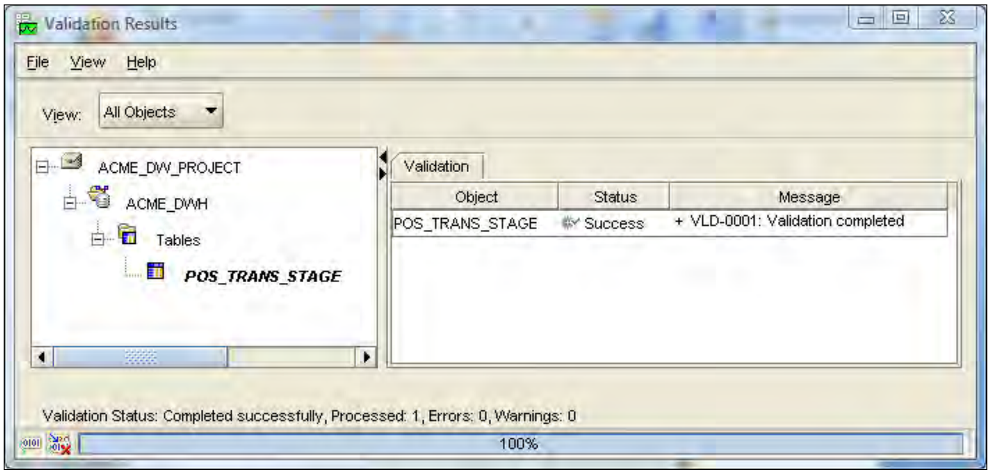
Validating in the Design Center

There is a context menu associated with everything we create. You can access it on any object in the Design Center by right-clicking on the object of your choice. Let's take a look at this by launching our Design Center, connecting as our **ACMEOWB** user, and then expanding our **ACME_DW_PROJECT**. Let's find our staging table, `POS_TRANS_STAGE`, and use it to illustrate the validation of an object from the Design Center. As we can recall, this table is under the **ACME_DWH** module in **Oracle node** and right-clicking on it will present us with the following pop-up menu:

Download at Boykma.Com



The **Validate...** entry has been highlighted. If we click on it, it will perform the validation of the metadata entered for the object and will present us with the results in a separate dialog box as shown next:



Object	Status	Message
POS_TRANS_STAGE	Success	+ VLD-0001: Validation completed

Validation Status: Completed successfully, Processed: 1, Errors: 0, Warnings: 0

100%

Download at Boykma.Com

This dialog box is resizable and has been made shorter to remove some blank space so that we can focus on the important parts. The window on the right will contain the messages that have resulted from the validation. Our `POS_TRANS_STAGE` table has validated successfully. But if we had any warnings or errors, they would appear in this window.

The validation will result in one of the following three possibilities:

1. The validation completes successfully with no warnings and/or errors as this one did.
2. The validation completes successfully, but with some non-fatal warnings.
3. The validation fails due to one or more errors having been found.

In any of these cases, we can look at the full message by double-clicking on the **Message** column in the window on the right. This will launch a small dialog box that contains the full text of the message.

The drop-down menu in the upper left has options for viewing all objects, just warnings, or just errors. The **All Objects** option, which is the default, displays all objects that have been validated, whether or not there were warnings or errors. We have the option to validate more than one object at a time by holding down the *Ctrl* key and clicking on several objects in the Design Center, and then with the *Ctrl* key still held down, right-clicking on one of them and then selecting **Validate**. All the selected objects will be validated and the results for all of them will appear in the window on the right. If we select **Warnings**, only the objects that have warnings will be displayed, and if we select **Errors**, only the objects with errors will be displayed.

If we have warnings or errors that we need to fix, we can double-click on the object name in the left window, or the name in the **Object** column in the right window, to launch the appropriate editor on the object—the Data Object Editor or the Mapping Editor. With one of these editors, we can make any modifications and revalidate the object. We won't have to go back to the Design Center to validate because the editors provide us the option to validate as well.

We can close the **Validation Results** dialog box now before moving on to discuss validating from the editors. We can click on **X** in the dialog box window header, or click on the **Close** entry from the **File** main menu, or use the *Ctrl+F4* key combination.

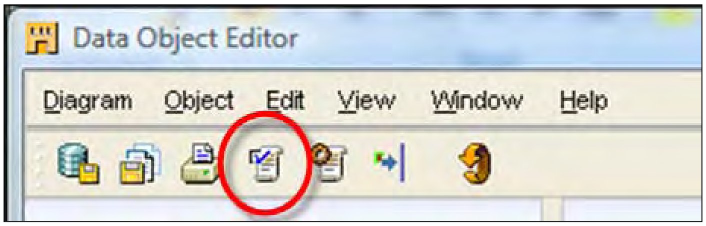
Validating from the editors

The Data Object Editor and the Mapping Editor both provide for validation from within the editor. We'll talk about the Data Object Editor first and then about the Mapping Editor because there are some subtle differences.

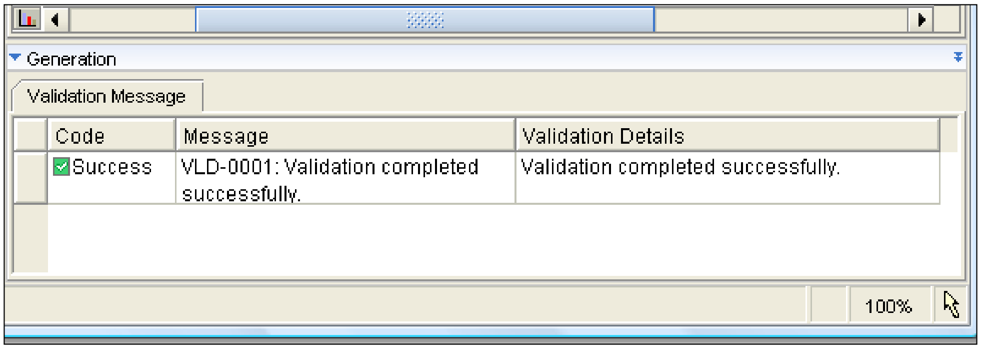
Download at Boykma.Com

Validating in the Data Object Editor

Let's double-click on the `POS_TRANS_STAGE` table name in the Design Center to launch the Data Object Editor so that we can discuss validation in the editor. We can right-click on the object displayed on the Canvas and select **Validate** from the pop-up menu, or we can select **Validate** from the **Object** menu on the main editor menu bar. Another option is available if we want to validate every object currently loaded into our Data Object Editor. It is to select **Validate All** from the **Diagram** menu entry on the main editor menu bar. We can also press the validate icon on the **General** toolbar, which is circled in the following image of the toolbar icons:



When we validate an object in the editor, we do not get the Validation Results pop-up dialog box as we did when validating from the Design Center. Here we get another window created in the editor, the **Generation** window, which appears below the **Canvas** window. This is what we used in the last chapter with our sneak peak at validation when we investigated the error we would get by connecting a date attribute to a number attribute. The window that is produced will look similar to the following:



Code	Message	Validation Details
✓ Success	VLD-0001: Validation completed successfully.	Validation completed successfully.

Download at Boykma.Com

This window will generally appear below the main Canvas window, but all these windows can be manipulated to our hearts' content. We can place them where we want and make them the size we want. The window can be collapsed so that all that is visible is the title bar that has the word **Generation** in it. The arrow to the left of the title can be used to open or close the window. It points down when the window is open, as seen in the above image. Clicking on it will cause the window to collapse and the arrow to rotate to point to the right, toward the window title. The two little down arrows on the right of the title bar are for a menu of options to manipulate the window, including maximizing the window, making the window full size to take up the entire area of the editor window, collapsing the window, or closing the window entirely. All the windows can be manipulated in this fashion. So, if a window doesn't appear where we want it to at first, we can move it around until it does.

In many cases, the error message will be long and the window will display the message truncated in the window. As we saw in the last chapter, we can double-click on the message and instead of creating a pop up with the full text of the message displayed, it makes the box containing the message expand until the entire message is visible.

When we validate from the Data Object Editor, it is on an object-by-object basis for objects appearing in the editor canvas. But when we validate a mapping in the Mapping editor, the mapping as a whole is validated all at once. Let's close the Data Object Editor and move on to discuss validating in the Mapping Editor.

Validating in the Mapping Editor

We'll now load our `STORE_MAP` mapping into the Mapping Editor and take a look at the validation options in that editor. Unlike the Data Object Editor, validation in the Mapping Editor involves all the objects appearing on the canvas together because that is what composes the mapping. We can only view one mapping at a time in the editor. The Data Object Editor lets us view a number of separate objects at the same time.

For this reason, we won't find a menu entry that says "Validate All" in the Mapping Editor. What we will find in the main editor menu bar's **Mapping** menu is an entry called **Validate** that we can select to validate this mapping. To the right of the **Validate** menu entry we can see the *F4* key listed, which means we can validate our mapping by pressing the *F4* key also. The key press we can do to validate is not available in the Data Object Editor because it wouldn't know which object we intend to validate. We also have a toolbar in the Mapping Editor that contains an icon for validation, like the one we have in the Data Object Editor. It is the same icon as shown in the previous image of the Data Object Editor toolbar.

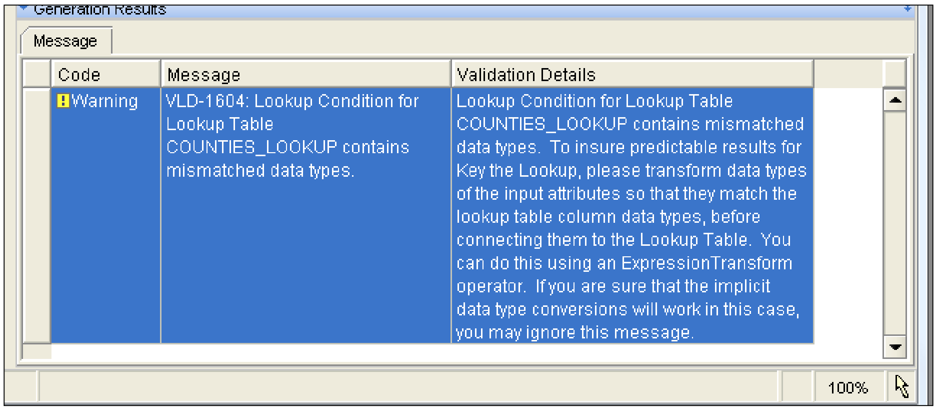
Download at Boykma.Com

Let's press the *F4* key or select **Validate** from the **Mapping** menu or press the Validate icon in the toolbar. The same behavior will occur as in the Data Object Editor—the validation results will appear in the **Generation Results** window.

A curious thing we'll find about validating in the editors is that the window title says **Generation** instead of **Validation**. As if this wasn't enough to confuse us, the window is labeled **Generation** in the Data Object Editor and **Generation Results** in the Mapping Editor. It's where both the validation and generation results will appear, but they contain extra information when generating as we'll see shortly. When validating in the Design Center, the pop-up results window is correctly labeled **Validation Results**.

As we'll soon see when we discuss the process of generation, it is closely tied to validation. Indeed, if we take a quick look at the pop-up window that appears when we validate an object in the Data Object Editor, one of the steps it performs is generating the object. The Generation window is fulfilling two purposes here, displaying the validation results and (as we'll see in a moment) displaying the results of the generation when that option is selected. The fact that the window name doesn't change is actually a minor bug that is an artifact from using the same window for both, which doesn't affect functionality at all. The only problem is the possible confusion for users. This should be corrected in some future version of the software.

We get a different result when validating the `STORE_MAP`. It does not report success, but gives us a warning. We will frequently encounter warnings from validation. They are non-fatal conditions that will not keep the object from being deployed or executed, but are items we should watch out for that might cause a problem. In many cases, we will be able to safely ignore the warnings. And in the case of the `STORE_MAP` warning, we can see that the message itself tells us we might be able to ignore the warning:



Code	Message	Validation Details
Warning	VLD-1604: Lookup Condition for Lookup Table COUNTIES_LOOKUP contains mismatched data types.	Lookup Condition for Lookup Table COUNTIES_LOOKUP contains mismatched data types. To insure predictable results for Key the Lookup, please transform data types of the input attributes so that they match the lookup table column data types, before connecting them to the Lookup Table. You can do this using an ExpressionTransform operator. If you are sure that the implicit data type conversions will work in this case, you may ignore this message.

Download at Boykma.Com

This message is telling us that it has compared the data type we supplied as input with the `COUNTIES_LOOKUP` table to the data type of the key field we're going to match against and found that they do not match. Let's take a look at them in the Mapping Editor and see what it's talking about.

In the Mapping Editor that has the `STORE_MAP` loaded, we'll click on the `VALUE` attribute in the `COUNTIES_LOOKUP` operator and then look at the **Attribute Properties** window on the left for the data type. We will see that it is defined as a **NUMBER** with zeros for precision and scale. When we click on the `ID` attribute in the output group of the `COUNTIES_LOOKUP` operator, we see in the **Properties** window that the data type of the `ID` attribute is the **INTEGER** type.

[Remember that the properties window is in the middle on the left side of the Mapping Editor interface, which is its default location. You may have to scroll the window down to find the attribute you're interested in.]

A NUMBER data type is a built-in data type in the Oracle Database. We can read all about data types in the *SQL Language Reference 11g Release 1* on the Oracle Technology web site `http://download.oracle.com/docs/cd/B28359_01/server.111/b28286/toc.htm`. This is discussed in this reference in Chapter 2, *Basic Elements of Oracle SQL*. The database uses these built-in data types internally to represent the information that can be stored in the database. For all numbers, data type would be the NUMBER data type. As we saw in the last chapter, precision and scale define how large a number is and how many decimal places are to the right of the decimal point.

There are several other terms used to represent data types that are common in other databases, programming languages, and tools. The Oracle database has provided us a feature that allows the database to recognize a large number of these alternative data types and will automatically convert them internally into one of its built-in types. These other supported data type representations are described in that same chapter from the language reference with tables provided to indicate the corresponding internal built-in data type. We can see that INTEGER is one such data type and is internally represented by NUMBER(38) as the equivalent.

So, we have a value we have said is a NUMBER data type being compared against a value we've said is an INTEGER data type. The Warehouse Builder validation process looks at the metadata representation, sees two different data types, and reports the warning even though we know an INTEGER is represented internally as a NUMBER. The problem is that a NUMBER can represent any kind of numeric value, not just an INTEGER, including floating-point numbers of any scale and precision.

Download at Boykma.Com

Our issue now is whether we can actually ignore the warning or not. We know that the actual numbers that will be used in this case are coming from a converted four-character string of digits, which we have extracted from the store number, and so will never have a decimal point (or scale). So, we are confident that the number will always be an integer. In that case there should be no issues with mapping the numbers, so we can safely ignore the warning.

> We could spend extra time to get rid of this warning so that we see the nice "Success" message instead. In this case, we would have to add an additional expression to manually convert the number returned by the `TO_NUMBER()` function into an integer before mapping it to the input group of the `COUNTIES_LOOKUP`. This might not be too difficult as the mapping is small and there is only one warning. But in most significant-sized mappings we're going to encounter, it quickly becomes too burdensome to try to get rid of every warning, and it would result in a much more complicated mapping.

That is validation in the mapping editor. We can go through our remaining objects and mappings now and validate them. The order we validate objects in is not critical. Unless we've made a typographical error, missed a selection we should have made, missed a column we should have added, or something like that, all the objects and mappings should validate successfully or have warnings that can be safely ignored. There will probably be a warning when validating the `STAGE_MAP` due to the `SALES_QUANTITY` from input being mapped to the `SALE_QUANTITY` in the `POS_TRANS_STAGE` table. The input has a precision of 22 set, and the output in the staging table is defined with a precision of zero. This can safely be ignored as a precision of zero will allow any size number to be stored up to the database maximum size.

> This process of validation only checks the logical design within the metadata in the Warehouse Builder. It can't check for any errors that might occur when the object is deployed and/or executed in the database. We haven't got to that point yet. We don't want to get the attitude that our objects and mappings are perfect just because they have passed validation. This is only the first step.
>
> The validations can even be misleading between objects. The `STORE` dimension would validate successfully if we removed the `DESCRIPTION` attribute that we talked about in Chapter 4. However, the `STORE_MAP` that used the dimension would give the error we talked about without the `DESCRIPTION` there to map to. So, validation is just one step along the way to getting a working data warehouse, but doesn't guarantee that there won't be further errors at a later point in the process.

When we are satisfied that everything is ready, we can move on to the generation step.

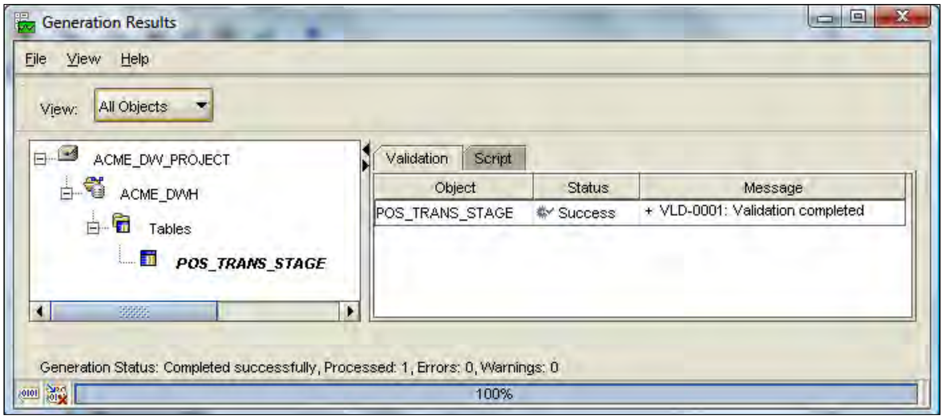
Download at Boykma.Com

Generating

This step can happen in conjunction with the validation step as we've discussed previously, but the Warehouse Builder does provide a separate menu entry to select for generating. We will discuss it here to see what it's all about. Let's talk about generation; and no, we're not talking about baby boomers, Gen X-ers, Gen Y-ers, or whatever they come up with for future generations. Here we're talking about the other meaning of the word, which is the act or process of generating. `Dictionary.com` says to generate means *to bring into existence; cause to be; produce*. With the generation step in the Warehouse Builder, we are going to bring into existence the code that we need to use to build and load our data warehouse. The objects—dimensions, cube, tables, and so on—will have SQL **Data Definition Language** (or **DDL**) statements produced, which when executed will build the objects in the database. The mappings will have the **PL/SQL** code produced that when it's run, will load the objects. The Warehouse Builder can also create configuration files for the **SQL*Loader** utility to load data or **ABAP** scripts, which are for interfacing to a **SAP** system. We're not going to make use of the SQL*Loader utility and we're not going to interface with a SAP system, so we won't need those options. We are going to use DDL and PL/SQL code for our project. More information about the other unused options is available in the *Warehouse Builder User's Guide*.

Generating in the Design Center

When we generate an object or mapping in the Design Center, we'll get the same pop-up window appearing as we got when we validated, but in this case the window will be labeled **Generation Results** and we'll have some extra information displayed. Let's go to the Design Center now and generate the code for the `POS_TRANS_STAGE` table. We'll right-click on it and select **Generate...** from the pop-up menu. It will present us with the following dialog box, which looks very similar to the one we got when we validated it:



Object	Status	Message
POS_TRANS_STAGE	Success	+ VLD-0001: Validation completed

Generation Status: Completed successfully, Processed: 1, Errors: 0, Warnings: 0

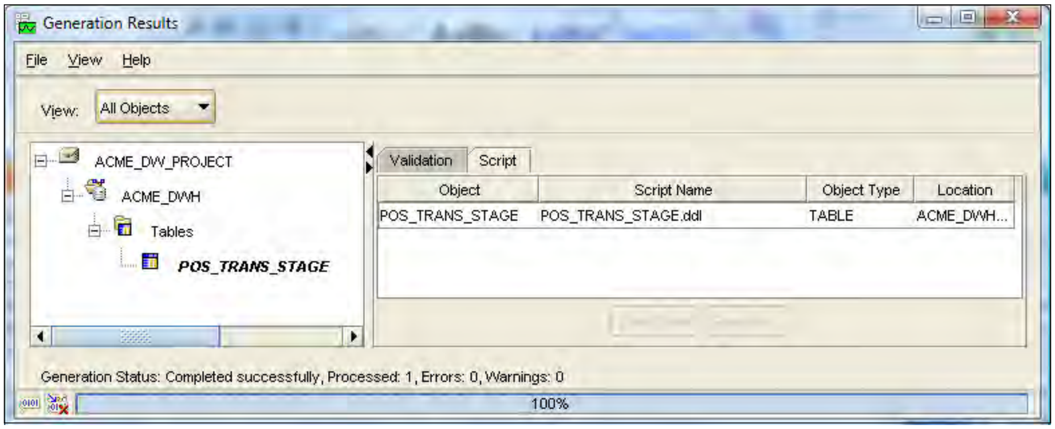
100%

Download at Boykma.Com

On the left is the list of the objects that we've generated; in this case only one appears. On the right is the window containing the results. If there were any messages too big to fit in the window, we could double-click on them to launch a pop-up window to display the full message just as we could with the validation results. These really are the validation results anyway. We get a validation for free when we do a generation.

We can see something extra this time that we didn't see when we just validated. There is a tab labeled **Script** that was not there before. This is where we can view the script that was generated, and which will create this object for us in the database. For a database object, a DDL script is generated for us. If we click on the **Script** tab, we can take a look at it.

Let's take a look at the **Script** tab. We see that it has generated a DDL script for us called `POS_TRANS_STAGE.ddl` as shown next:



Object	Script Name	Object Type	Location
POS_TRANS_STAGE	POS_TRANS_STAGE.ddl	TABLE	ACME_DWH...

Generation Status: Completed successfully, Processed: 1, Errors: 0, Warnings: 0

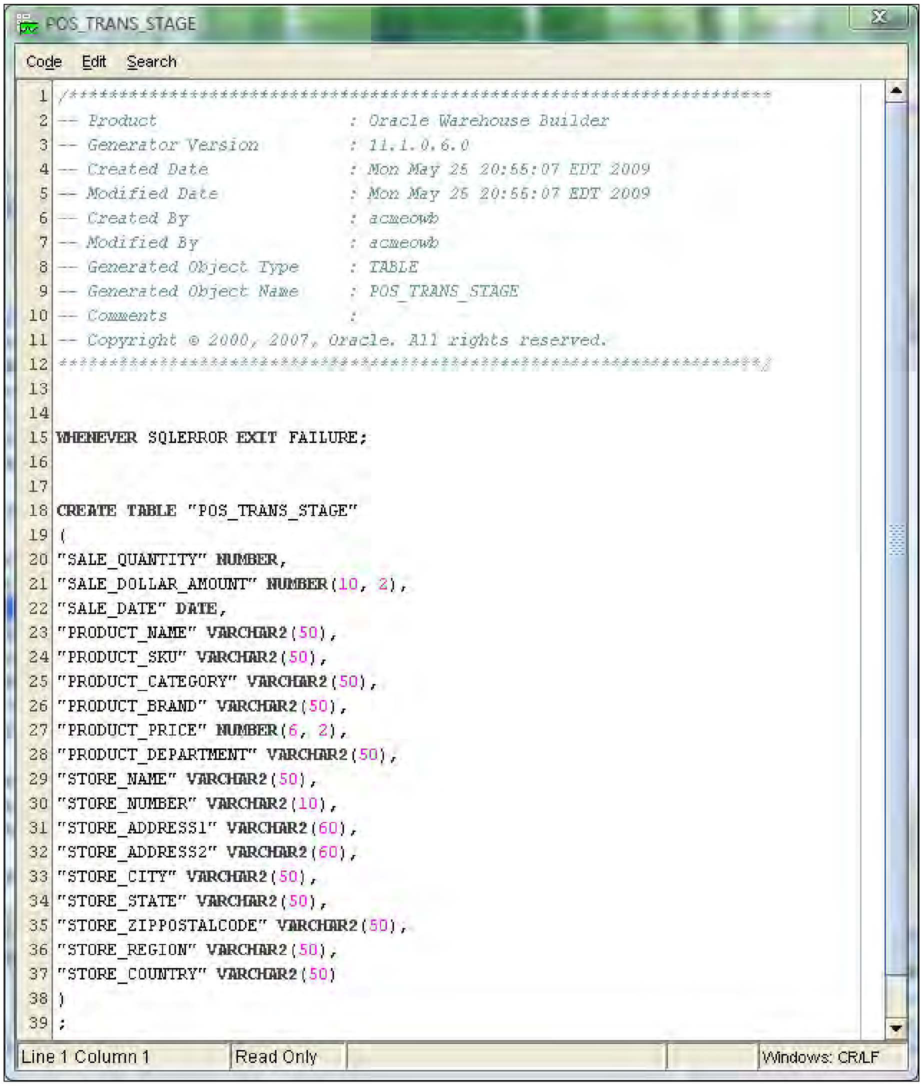
100%

We have two options in this dialog box:

- We can view the script
- We can save it to the disk

Download at Boykma.Com

Let's click on the script name, or any one of the columns on that line, and we'll see that the **View Code** and **Save As...** buttons have become active; they are no longer grayed out. We'll click the **View Code** button and will be presented with a dialog box displaying the code shown next:



```
/***********************************************************************
-- Product                      : Oracle Warehouse Builder
-- Generator Version            : 11.1.0.6.0
-- Created Date                 : Mon May 25 20:55:07 EDT 2009
-- Modified Date                : Mon May 25 20:55:07 EDT 2009
-- Created By                   : acmeowb
-- Modified By                  : acmeowb
-- Generated Object Type        : TABLE
-- Generated Object Name        : POS_TRANS_STAGE
-- Comments                     :
-- Copyright © 2000, 2007, Oracle. All rights reserved.
***********************************************************************/


WHENEVER SQLERROR EXIT FAILURE;


CREATE TABLE "POS_TRANS_STAGE"
(
"SALE_QUANTITY" NUMBER,
"SALE_DOLLAR_AMOUNT" NUMBER(10, 2),
"SALE_DATE" DATE,
"PRODUCT_NAME" VARCHAR2(50),
"PRODUCT_SKU" VARCHAR2(50),
"PRODUCT_CATEGORY" VARCHAR2(50),
"PRODUCT_BRAND" VARCHAR2(50),
"PRODUCT_PRICE" NUMBER(6, 2),
"PRODUCT_DEPARTMENT" VARCHAR2(50),
"STORE_NAME" VARCHAR2(50),
"STORE_NUMBER" VARCHAR2(10),
"STORE_ADDRESS1" VARCHAR2(60),
"STORE_ADDRESS2" VARCHAR2(60),
"STORE_CITY" VARCHAR2(50),
"STORE_STATE" VARCHAR2(50),
"STORE_ZIPPOSTALCODE" VARCHAR2(50),
"STORE_REGION" VARCHAR2(50),
"STORE_COUNTRY" VARCHAR2(50)
)
;
```

We can see that it has generated an SQL `CREATE TABLE` statement for us. It contains the name of our `POS_TRANS_STAGE` table along with column names and types as they are defined in the Warehouse Builder. The dialog box provides us a menu bar with three entries—**Code**, **Edit,** and **Search**—which we can use to do tasks such as:

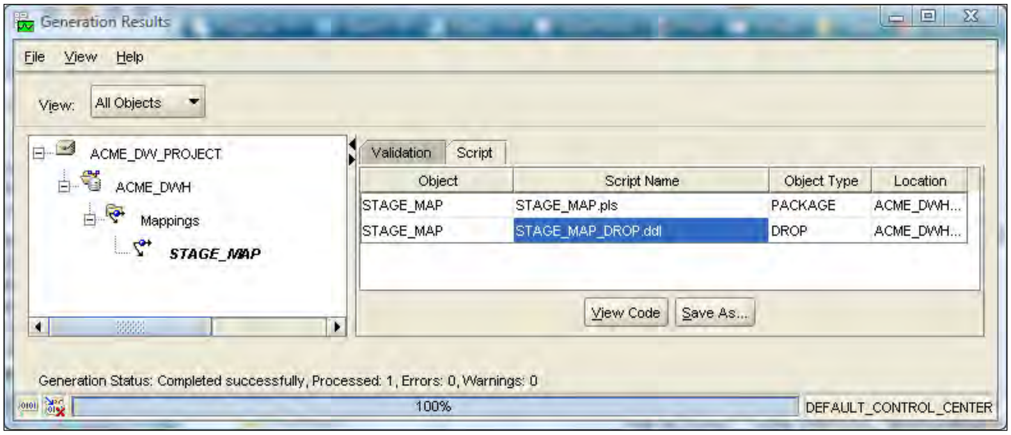
Download at Boykma.Com

- Saving this code to a file
- Copying portions of the code and pasting them into another window
- Searching through the code for text strings

However, we can see that the script is displayed in a read-only mode. In other words, we are not allowed to make any changes to it. It is code that is automatically generated by the Warehouse Builder, so there is no way for us to edit it directly.

To deploy our objects and mappings to create and populate our data warehouse, we really need not be concerned with what the code looks like. This is because the Warehouse Builder takes care of generating it all for us. However, we're checking this out for the first time to get an appreciation of what it is doing for us behind the scenes. The data object code is not complicated in this case, but let's take a look at the code for the mapping to populate this data object. We can close out the code viewing dialog box and the **Generation Results** dialog box for the `POS_TRANS_STAGE` table.

We'll right-click on the `STAGE_MAP` mapping in the Design Center and select **Generate** from the pop-up menu. There may be something different that is noticeable on the **Script** tab of the **Generation Results** window, depending on whether we've generated and actually deployed this mapping before:



Object	Script Name	Object Type	Location
STAGE_MAP	STAGE_MAP.pls	PACKAGE	ACME_DWH...
STAGE_MAP	STAGE_MAP_DROP.ddl	DROP	ACME_DWH...

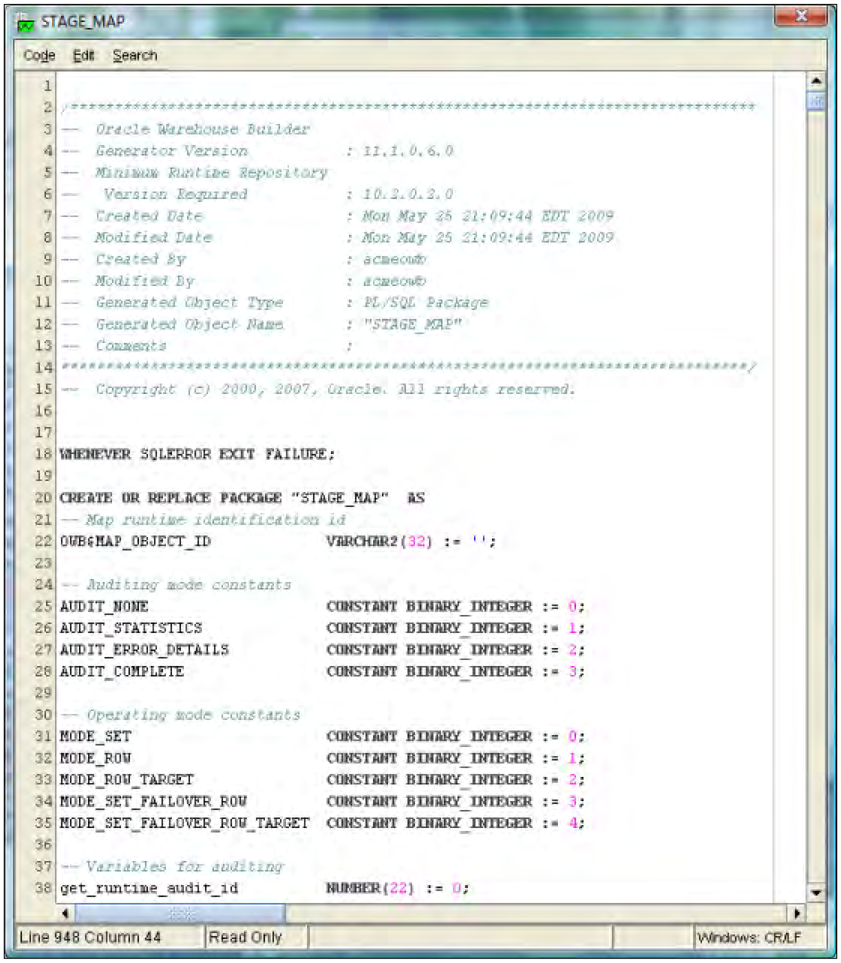
Generation Status: Completed successfully, Processed: 1, Errors: 0, Warnings: 0

100%

There may be an extra script that gets generated for us here to do a drop of the object first. If we've deployed the mapping previously and have now made some edits to it to regenerate and redeploy, it defaults to a replace option for the deploy action and creates the script to drop the mapping along with any temporary tables that got generated to support the execution of the mapping.

Download at Boykma.Com

Whether we've deployed the mapping or not, we still get the code generated that will create the mapping code for us in the database. This is the `STAGE_MAP.pls` package that appears in the window. We'll discuss the deployment actions a little later in the chapter. If we view this code, we'll see that it is more complicated than the DDL script that was generated for our previous table. Let's click on the `STAGE_MAP.pls` line and then on the **View Code** button to view the code. The same pop-up window is used to view this mapping code as we saw before when we viewed the DDL script for the table. However, the code is quite different as it is PL/SQL code as shown in the following image:



```

/*****************************************************************************
--  Oracle Warehouse Builder
--  Generator Version           : 11.1.0.6.0
--  Minimum Runtime Repository
--   Version Required           : 10.2.0.2.0
--  Created Date                : Mon May 25 21:09:44 EDT 2009
--  Modified Date               : Mon May 25 21:09:44 EDT 2009
--  Created By                  : acmeowb
--  Modified By                 : acmeowb
--  Generated Object Type       : PL/SQL Package
--  Generated Object Name       : "STAGE_MAP"
--  Comments                    :
*****************************************************************************/
--  Copyright (c) 2000, 2007, Oracle. All rights reserved.


WHENEVER SQLERROR EXIT FAILURE;

CREATE OR REPLACE PACKAGE "STAGE_MAP"  AS
-- Map runtime identification id
OWB$MAP_OBJECT_ID              VARCHAR2(32) := '';

-- Auditing mode constants
AUDIT_NONE                     CONSTANT BINARY_INTEGER := 0;
AUDIT_STATISTICS               CONSTANT BINARY_INTEGER := 1;
AUDIT_ERROR_DETAILS            CONSTANT BINARY_INTEGER := 2;
AUDIT_COMPLETE                 CONSTANT BINARY_INTEGER := 3;

-- Operating mode constants
MODE_SET                       CONSTANT BINARY_INTEGER := 0;
MODE_ROW                       CONSTANT BINARY_INTEGER := 1;
MODE_ROW_TARGET                CONSTANT BINARY_INTEGER := 2;
MODE_SET_FAILOVER_ROW          CONSTANT BINARY_INTEGER := 3;
MODE_SET_FAILOVER_ROW_TARGET   CONSTANT BINARY_INTEGER := 4;

-- Variables for auditing
get_runtime_audit_id           NUMBER(22) := 0;
```

Download at Boykma.Com

For a mapping, the code that is generated is a PL/SQL **Package**. A package is a code construct that contains variables and procedures to perform some work in the database. It is a way to group variables and procedures together that all contribute to the performance of a particular task. In this case, the task is the loading of the `POS_TRANS_STAGE` table. All the code necessary to accomplish that task is contained in this package's script. We have the same **Code**, **Edit**, and **Search** menu options and the script is also read-only, as it was for the table object we displayed previously. If we scroll this window all the way to the bottom, we see that the Warehouse Builder has generated about 4,500 lines of code. This is much more than it generated for the table object previously, and makes us real glad we didn't have to write all that code ourselves.

This completes our look at the process we would follow to generate our objects and mappings from the Design Center. If we were to encounter any errors that needed to be fixed, we could jump to the appropriate editor by double-clicking on the object or mapping name in the left window of the results dialog box.

As with validation, the generation function can be run from the editors as well and this is slightly different from running it from the Design Center. So let's take a look at that now after closing out the script window and the generation results dialog box.

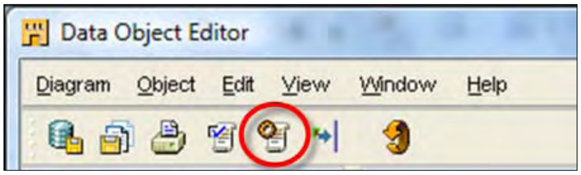
Generating from the editors

When we generate an object or mapping from one of the editors, it will display the results in a window within the editor just as it did for the validation; only this time the name of the window will match the function we just performed. The same **Generation** window in which the validation results appeared will be used for the actual generation results. But as with the generation from the Design Center, we'll have the additional information available. The procedure for generating from the editors is the same as for validation, but the contents of the results window will be slightly different depending on whether we're in the Data Object Editor or the Mapping Editor. Let's discuss each individually as we previously did.

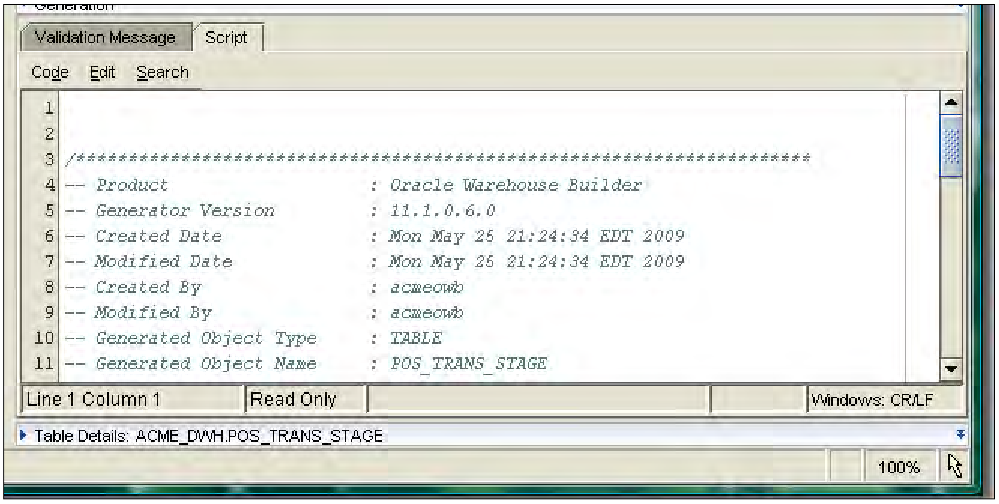
Download at Boykma.Com

Generating in the Data Object Editor

We'll start with the Data Object Editor and open our `POS_TRANS_STAGE` table in the editor by double-clicking on it in the Design Center. To review the options we have for generating, there is the **Generate...** menu entry under the **Object** main menu, the **Generate** entry on the pop-up menu when we right-click on an object, and a **Generate** icon on the general toolbar right next to the **Validate** icon as shown in the following image:



We'll use one of these methods to generate the `POS_TRANS_STAGE` table. The results will appear in the following **Generation** window with the **Script** tab selected:



```
/**********************************************************************
-- Product                       : Oracle Warehouse Builder
-- Generator Version             : 11.1.0.6.0
-- Created Date                  : Mon May 25 21:24:34 EDT 2009
-- Modified Date                 : Mon May 25 21:24:34 EDT 2009
-- Created By                    : acmeowb
-- Modified By                   : acmeowb
-- Generated Object Type         : TABLE
-- Generated Object Name         : POS_TRANS_STAGE
```

Line 1 Column 1 | Read Only | Windows: CR/LF

Table Details: ACME_DWH.POS_TRANS_STAGE

100%

The window also provides us a **Validation Messages** tab, which will display any messages generated as a result of validation. We also have a menu of options that appears on the **Script** tab: **Code**, **Edit**, and **Search** that provide us the same options as on the pop-up window when viewing the script from the Design Center that we saw previously.

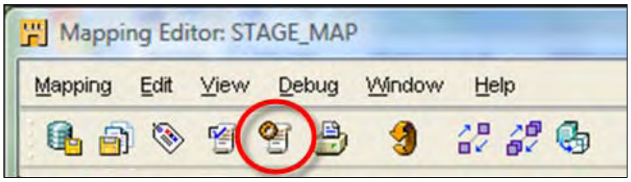
Download at Boykma.Com

Even though it says **Edit**, don't be fooled. We are still not allowed to edit the script that displays in the window; it is read-only. This is a good thing anyway, as we are using the Warehouse Builder precisely to make our lives easier by not having to worry about writing or editing code. We can do things such as save the script to a file, copy portions of it and paste it into another application, or search through the script for text.

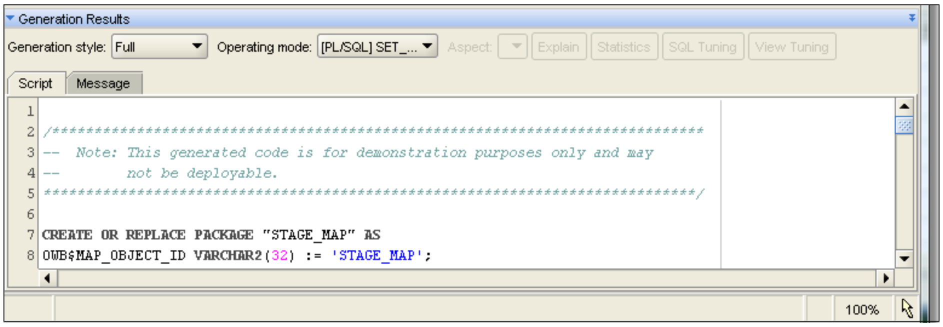
This is it basically. The script that displays is the same one that we saw when generating in the Design Center. Now let's take a look at generating in the Mapping Editor because the generation results window will look a little different as it's a mapping we'll generate and not a data object.

Generating in the Mapping Editor

Let's open the Mapping Editor on the `STAGE_MAP` mapping now. From the Design Center, we can just double-click on the `STAGE_MAP` mapping to launch the Mapping Editor and load the `STAGE_MAP` mapping. The process for generating the mapping is the same as for validation, but we just select **Generate** instead of Validate from the **Mapping** menu or press the *F5* key instead of *F4*. There is also an icon on the toolbar to select the generation option. It is shown next:



Looking at the **Generation** window now, we can see that we have additional information displayed. Instead of the validation messages appearing in the window, we now see the script that was generated. We also have a couple of extra drop-down menus, which we didn't have when we were just validating, and which we didn't have when we were generating in the Data Object Editor either. An example of what we'll see is shown next:



```
/***************************************************************************
--  Note: This generated code is for demonstration purposes only and may
--        not be deployable.
***************************************************************************/

CREATE OR REPLACE PACKAGE "STAGE_MAP" AS
OWB$MAP_OBJECT_ID VARCHAR2(32) := 'STAGE_MAP';
```

Download at Boykma.Com

We have two tabs available in the window, a **Script** tab that displays the script and a **Message** tab that displays the validation messages. These perform the same functions as the **Script** and **Validation Message** tabs in the **Generation** window of the Data Object Editor. However, the Script tab doesn't have the extra Code, Edit, and Search menus that we had in the Data Object Editor. The reason can be found in this comment we see in the script window as a note:

```
/*********************************************************************
*****************
-- Note: This generated code is for demonstration purposes only and may
--       not be deployable.
**********************************************************************
****************/
```

The code generated for a mapping is far more complex than the DDL code generated for data objects as we saw when we looked at it from the Design Center. One of the main reasons the code for mappings is so complex is because we have five options to choose from for the **default operating mode** of the mapping when we execute it, and it has to be able to support all five. There are three operating modes the mapping can run in, and two that indicate failover options for switching between them. One of the drop-down menus at the top of the **Script** tab window is labeled **Operating mode**, and allows us to view the code for each of the operating modes separately. For this reason, some different functionality is available to us in the **Script** tab and it displays code that is not quite the complete script we saw in the Design Center. If we scroll down the **Script** tab, we'll see that it is not displaying as many lines as were displayed when viewing the generation results from the Design Center. That is what the note we mentioned previously is referring to. Viewing the script from the Design Center shows us the complete script, which includes support for the code of all operating modes.

Default operating mode of the mapping

The Warehouse Builder provides three possible modes that the mapping code can run in when executing in the database. In addition, it also provides two options for failover execution should one mode have errors. These modes are based on the performance we expect from the mapping, the amount of auditing data we require, and how we designed the mapping. The *Warehouse Builder User's Guide* has a much more detailed discussion of this topic in Chapter 22—*Understanding Performance and Advanced ETL Concepts*. For more advanced mappings, we'll definitely want to be familiar with concepts from that chapter. But for our purpose here, we'll only cover a very high-level view with just enough information to explain the different options we have in the Script tab of the Mapping Editor for viewing the code.

Download at Boykma.Com

The three modes are as follows:

- Set-based
- Row-based
- Row-based (target only)

In set-based mode, the Warehouse Builder will generate a single SQL statement that performs all the operations of our mapping in one statement. It processes the data as a single set of data. This is good for performance, but the drawback is that runtime auditing information is limited. If any errors are generated, it is not able to tell us which row generated the error. We can view the code that is set-based by selecting **SET_BASED** in the **Script** tab from the **Operating Mode** drop-down menu.

In row-based mode, the Warehouse Builder generates code to process the data row by row. It uses a combination of SQL Cursors and PL/SQL code. It does not provide as big a performance benefit as the set-based mode, but we gain much greater auditing capability of the execution results. There are also additional parameters that can be set to improve the performance of this mode, which are documented in the User's Guide and online help if we choose to use this mode. We can select **ROW_BASED** from the drop-down menu to view the row-based code.

The final of the three options is row-based (target only) mode. This option creates a SQL select cursor and tries to include as many operations as it can into that cursor to process the source data and operations on it as a set, but then writes the rows to the target one row at a time. This will limit the auditing available for input and operations, but provides greater auditing of the output to the target. We can select **ROW_BASED_TARGET_ONLY** from the drop-down menu to view the code for the option.

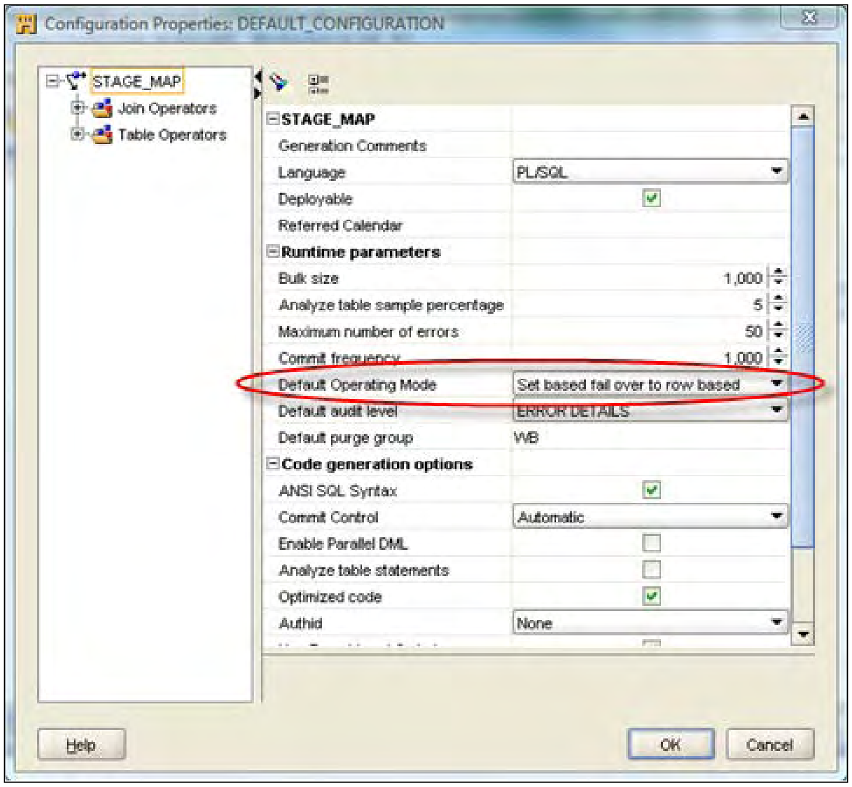
The following two additional options for operating modes are available, which are based on the previous three:

- Set-based fail over to row-based
- Set-based fail over to row-based (target only)

These options are used to run the mapping in set-based mode, but if an error occurs, try the mapping in row-based mode—either regular or target only. We can view the code for either of these options by selecting **SET_BASED_FAIL_OVER_TO_ROW_BASED** or **SET_BASED_FAIL_OVER_TO_ROW_BASED_TARGET_ONLY** from the drop-down menu.

Download at Boykma.Com

Changing that drop-down menu does not change the actual mode that will be used to run the mapping. For that we have to view the configuration options for the mapping from the Design Center. The options are available via right-clicking a mapping in the Design Center and selecting **Configure...**. This will display the following pop-up screen where we can see that the default operating mode set for our mapping is **Set based fail over to row based**. This is the default that is set for all mappings, as shown next:



STAGE_MAP
Generation Comments
Language: PL/SQL
Deployable: ✓
Referred Calendar
Runtime parameters
Bulk size: 1,000
Analyze table sample percentage: 5
Maximum number of errors: 50
Commit frequency: 1,000
Default Operating Mode: Set based fail over to row based
Default audit level: ERROR DETAILS
Default purge group: WB
Code generation options
ANSI SQL Syntax: ✓
Commit Control: Automatic
Enable Parallel DML: ☐
Analyze table statements: ☐
Optimized code: ✓
Authid: None

We will not have to modify any of these options for our mappings and so will not spend any more time with this dialog box. More information about these options can be found in the *Warehouse Builder User's Guide*. We'll move on to discuss the other drop-down menu that appears in the **Script** tab of the **Mapping Editor Generation Results** window, the **Generation style** drop-down menu.

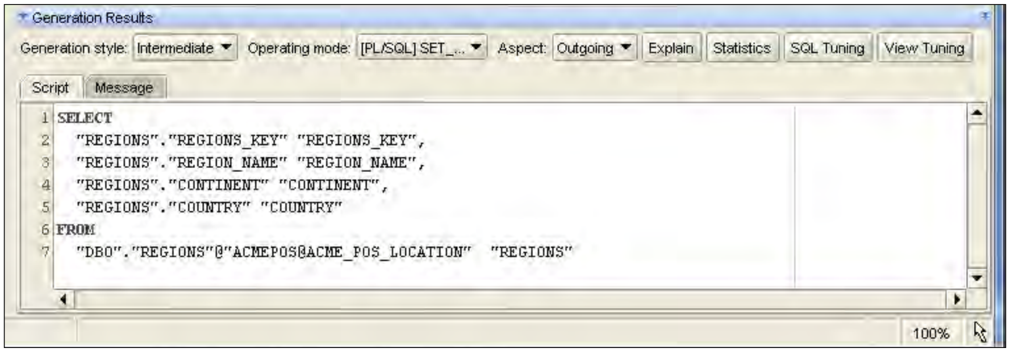
Download at Boykma.Com

Selecting the generation style

The generation style has two options we can choose from, **Full** or **Intermediate**. The Full option will display the code for all operators in the complete mapping for the operating mode selected. The Intermediate option allows us to investigate code for subsets of the full mapping option. It displays code at the attribute group level of an individual operator. If no attribute group is selected when we select the intermediate option in the drop-down menu, we'll immediately get a message in the **Script** tab saying the following:

```
Please select an attribute group.
```

When we click on an attribute group in any operator on the mapping, the Script window immediately displays the code for setting the values of that attribute group. The following is an example of what we would see by clicking on the **INOUTGRP1** group of the **REGIONS** table operator:



```
SELECT
  "REGIONS"."REGIONS_KEY" "REGIONS_KEY",
  "REGIONS"."REGION_NAME" "REGION_NAME",
  "REGIONS"."CONTINENT" "CONTINENT",
  "REGIONS"."COUNTRY" "COUNTRY"
FROM
  "DBO"."REGIONS"@"ACMEPOS@ACME_POS_LOCATION"  "REGIONS"
```

It is a standard SQL `SELECT` statement that has the four attributes of the `REGIONS` table selected. The `FROM` clause indicates that the source of the data for these attributes is the `REGIONS` table in the `ACME_POS` database at our location defined as `ACME_POS_LOCATION`. In Chapter 2, we discussed the non-Oracle Database module we created for the ACME POS transactional database. We called it `ACME_POS` with a location defined as `ACME_POS_LOCATION`. The Warehouse Builder implements this location as a database link in the Oracle Database and calls it `ACMEPOS@ACME_POS_LOCATION`. The `ACMEPOS` text string is from the service name we used to refer to the location and `ACME_POS_LOCATION` is the name we gave to the location. To reference a table in that database, the table name is prefixed to the database link name, which is separated by another `@` symbol. To further specify the exact table, the schema name is prefixed to the table name separated by a period. Looking back at Chapter 2 where we defined the `ACME_POS_LOCATION`, we specified `DBO` as the schema name in the **Edit non-Oracle Location** dialog box along with `ACMEPOS` as the service name and `ACME_POS_LOCATION` as the location name. This is where all that information used in the previous script came from.

Download at Boykma.Com

When we selected the **Intermediate generation** style, the drop-down menu and buttons on the righthand side of the window became active. We have a number of options for further investigation of the code that is generated, but these are beyond the scope of what we'll be covering in this book. One final point we'll make about these options is in reference to the drop-down menu labeled **Aspect**. When an attribute group is selected, it may be used as input, output, or both. This drop-down menu lets us see the code that is defined for either one of these options for an attribute group that has more than one of these options. If we select **Incoming**, we get to see what code selects the values that are used as input for the group. If we select **Outgoing**, we get to see the code that will select the values for output. There is another option that can appear in the menu, and that is **Loading**. If we click on the `INOUTGRP1` attribute group of the `POS_TRANS_STAGE` mapping table operator, we'll have that option in addition to **Incoming** and **Outgoing**. This is the final target operator in the mapping, and so must actually load data into the target table. The loading aspect will show us the SQL `INSERT` statement that loads the data into the target table.

This concludes our discussion about generating code for mappings and objects. So let's close the Mapping Editor and proceed to the next step, which is to deploy our objects and mappings to the database.

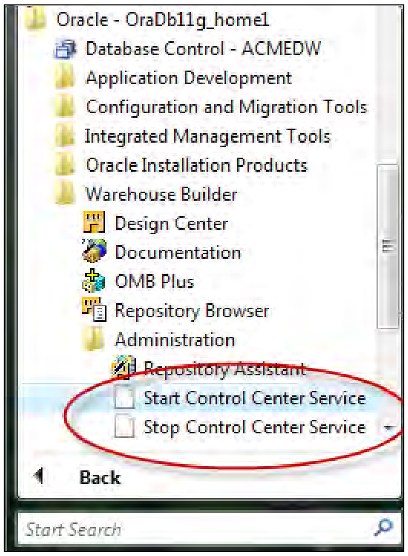
Deploying

The process of deploying is where database objects are actually created and PL/SQL code is actually loaded and compiled in the target database. Up until this point, no objects exist in our target schema, `ACME_DWH`, in our Oracle database. Everything we've talked about so far about importing metadata for tables, defining objects, mappings, and so on has referred totally to the Warehouse Builder repository, where it keeps a record of everything we've defined so far in metadata. Not a single actual database object has been created yet. Everything we've done until now has been done entirely in the OWB Design Center client. But to perform the process of deployment, now we're going to have to communicate to the target database. For that we need to be introduced to the **Control Center Service**, which must be running for the deployments to function.

Download at Boykma.Com

The Control Center Service

If we briefly look all the way back at Chapter 1, we talked about the architecture of the Warehouse Builder and looked at a diagram that laid out the main components of the Warehouse Builder software and where they were located—either on the client or on the server. The Control Center Service is a process that runs on the server and provides the interface to our target database for controlling the deployment process. It is also possible to run the Control Center Service on another remote computer to implement a remote runtime. If it's running in this configuration, it doesn't start automatically by default. So we would need to manually start it. It is available from the Windows **Start** menu as shown next:


Oracle - OraDb11g_home1
Database Control - ACMEDW
Application Development
Configuration and Migration Tools
Integrated Management Tools
Oracle Installation Products
Warehouse Builder
Design Center
Documentation
OMB Plus
Repository Browser
Administration
Repository Assistant
Start Control Center Service
Stop Control Center Service
Back
Start Search

We will not need to run it because we're running locally on the same machine as the database is running, and will be interfacing with the Control Center Service that is running locally in the database. If we were to implement a remote runtime and had to run this **Start Control Center Service** menu entry, it would start up a command window with the window title Start Control Center Service and would pop-up a dialog box asking us for connection information for the OWBSYS schema in our database. We would enter the password, host name, and service name for connecting to that schema.

Download at Boykma.Com

The local Control Center Manager on the database server is controlled using scripts, which are run in the database while connected as the OWBSYS user. The scripts are located in the `ORACLE_HOME\owb\rtp\sql\` folder. They can be run using the SQL*Plus command-line utility for executing SQL commands and scripts. Open a command-prompt window and enter the following command to run it and connect to the OWBSYS schema:

```
sqlplus OWBSYS
```

Enter the password for OWBSYS when prompted, and then enter the following command at the SQL*Plus command prompt to display the status of the service:

```
@ORACLE_HOME\owb\rtp\sql\show_service.sql
```

Substitute your actual ORACLE_HOME location in the previous command.

A few of the other scripts available in the previous folder are as follows:

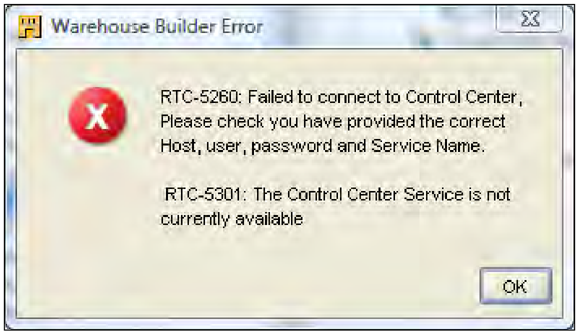
- `start_service.sql`: Starts the Control Center Service
- `stop_service.sql`: Stops the Control Center Service
- `service_doctor.sql`: Analyzes the state of the service and reports the status

The Control Center Service normally starts when the database starts up. So if we are running the database server locally, we don't need to bother with running any of the scripts. However, it is good to be informed should there be any problem in the future involving connections to the service. Let's give the Control Center Service some work to do now by doing an actual deployment from the Design Center.

Deploying in the Design Center and Data Object Editor

As with validation and generation, we can deploy objects and mappings from the Design Center or from within the Data Object Editor. However, unlike validation and generation, there is no deployment option available from the Mapping Editor. Let's deploy our `POS_TRANS_STAGE` table from the Design Center. The process is similar if we're doing it from the Data Object Editor. We'll right-click on it and select **Deploy** from the pop-up menu. If the Control Center Service is not running for some reason, we'll be presented with an error dialog box as shown next:

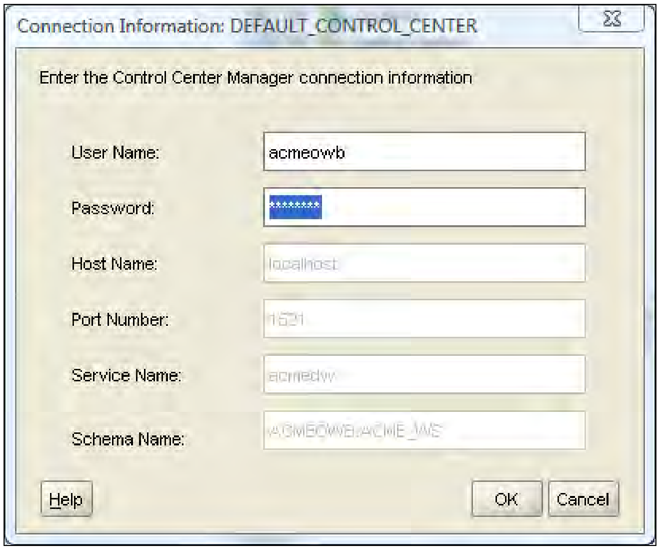
Download at Boykma.Com


Warehouse Builder Error

RTC-5260: Failed to connect to Control Center, Please check you have provided the correct Host, user, password and Service Name.

RTC-5301: The Control Center Service is not currently available

If we get this pop-up window when we click on the **OK** button, we'll get another pop-up window prompting for connection information for the Control Center Service where we can provide connection information. That dialog box looks like the following:


Connection Information: DEFAULT_CONTROL_CENTER

Enter the Control Center Manager connection information

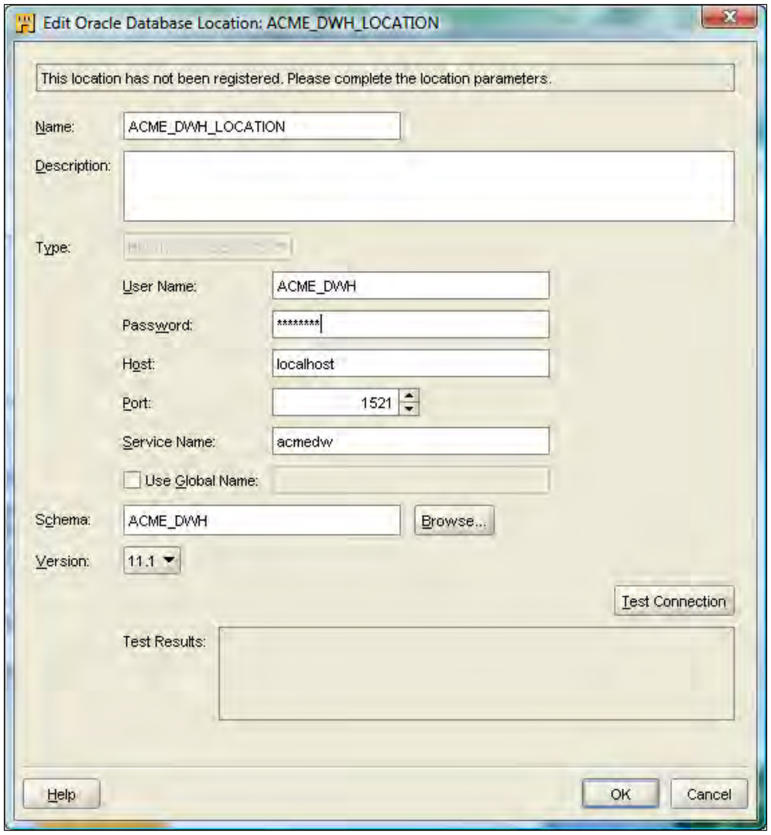
User Name: acmeowb
Password: ********
Host Name: localhost
Port Number: 1521
Service Name: acmedw
Schema Name: ACMEOWB.ACME_WS

The only items we can modify here are the **User Name** and **Password** to use. These items default to the repository workspace owner that we've been using all along to connect in the Design Center, and we should not change them. Mostly, this dialog box appears because the Control Center Service is not running, and not because of incorrect connection information. Let's take a quick look at where that connection information is specified. We'll press the **Cancel** button in the dialog box to close it if it appears.

Download at Boykma.Com

The Control Center Service connection information is set in the **Design Center**, which is in the **Connection Explorer** window under the **Control Centers** entry. If we expand it, we can see that a default Control Center Service was created for us called `DEFAULT_CONTROL_CENTER`. We did not have to create it separately. If we double-click on this, we get a dialog box that shows the connection information. However, we're not able to edit anything in it. When this entry was created it was specified, and it can't be changed. Therefore, it is unlikely that incorrect connection information caused the Control Center Service error dialog box to appear. We could have more than one control center defined. But the default will work fine for us, so we will not modify it.

If our Control Center Service is running, which will usually be the case, we won't get the previous dialog boxes. However, we may get the following dialog box telling us that our location is not registered:


Edit Oracle Database Location: ACME_DWH_LOCATION

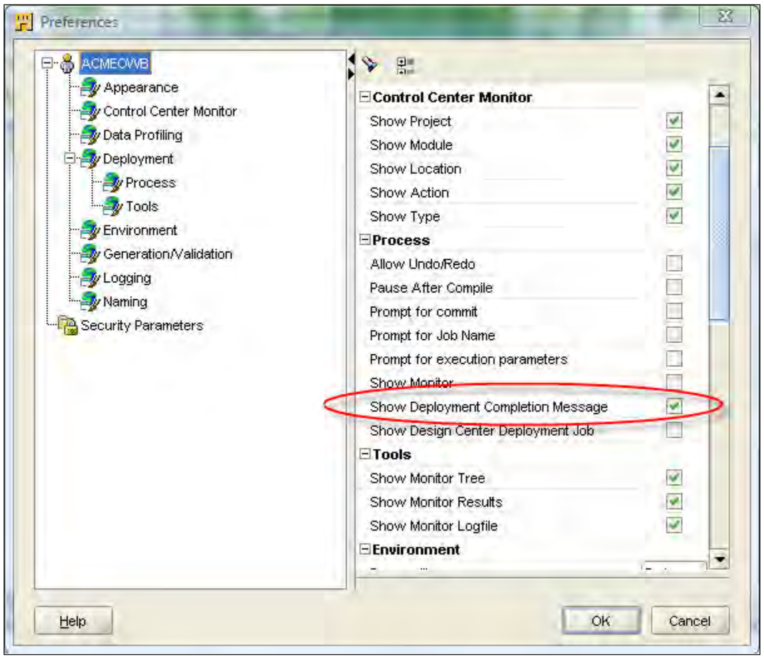
This location has not been registered. Please complete the location parameters.

Name: ACME_DWH_LOCATION
Description:
Type:
User Name: ACME_DWH
Password: ********
Host: localhost
Port: 1521
Service Name: acmedw
Use Global Name:
Schema: ACME_DWH
Version: 11.1

Download at Boykma.Com

The act of registering a location is to associate all the previous information with a location defined in the Warehouse Builder so that the Control Center Service knows how to find the location. The connection details in the previous dialog box are what it uses to connect to the location. Simply provide the password for the target schema and ensure that the rest of the information is correct, and then click on **OK**. The location will be registered and the object will deploy. We can register and un-register locations at any time by using the Control Center Manager. We'll be looking at the Control Center Manager very soon.

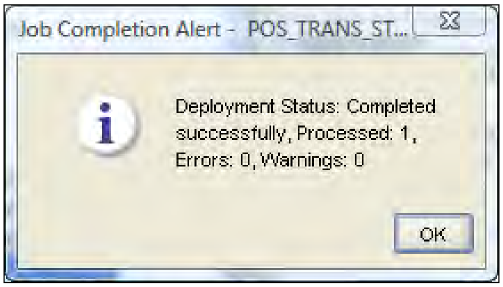
So, we have deployed our `POS_TRANS_STAGE` table in the Design Center. Assuming the Control Center Service is running and we don't get any of the previous dialog boxes, we will actually not get any dialog box when we deploy an object, unless we have set a certain option to tell the Warehouse Builder to display a dialog box upon completion of the deployment. If this option is not set (which is the default), it will show some messages in the message bar at the bottom of the window as it's doing the deployment. But it will not tell us whether the deployment succeeded or failed. If we want to see a completion message, we have to set this option in the preferences for the Design Center and tell it to show this message to us when the deployment is completed. We can set this option under the **Tools** menu entry by selecting the **Preferences...** menu entry. The resulting dialog box will look similar to the following, which has been scrolled down so that the needed entry is visible and the column has been expanded so that the whole description is visible. We need to check the box beside the option for displaying the completion status as follows:



Control Center Monitor	
Show Project	✓
Show Module	✓
Show Location	✓
Show Action	✓
Show Type	✓
Process	
Allow Undo/Redo	☐
Pause After Compile	☐
Prompt for commit	☐
Prompt for Job Name	☐
Prompt for execution parameters	☐
Show Monitor	☐
Show Deployment Completion Message	✓
Show Design Center Deployment Job	☐
Tools	
Show Monitor Tree	✓
Show Monitor Results	✓
Show Monitor Logfile	✓
Environment	

Download at Boykma.Com

It will show us the completion message after this option has been checked, which will look similar to the following:



Deployment Status: Completed successfully, Processed: 1, Errors: 0, Warnings: 0

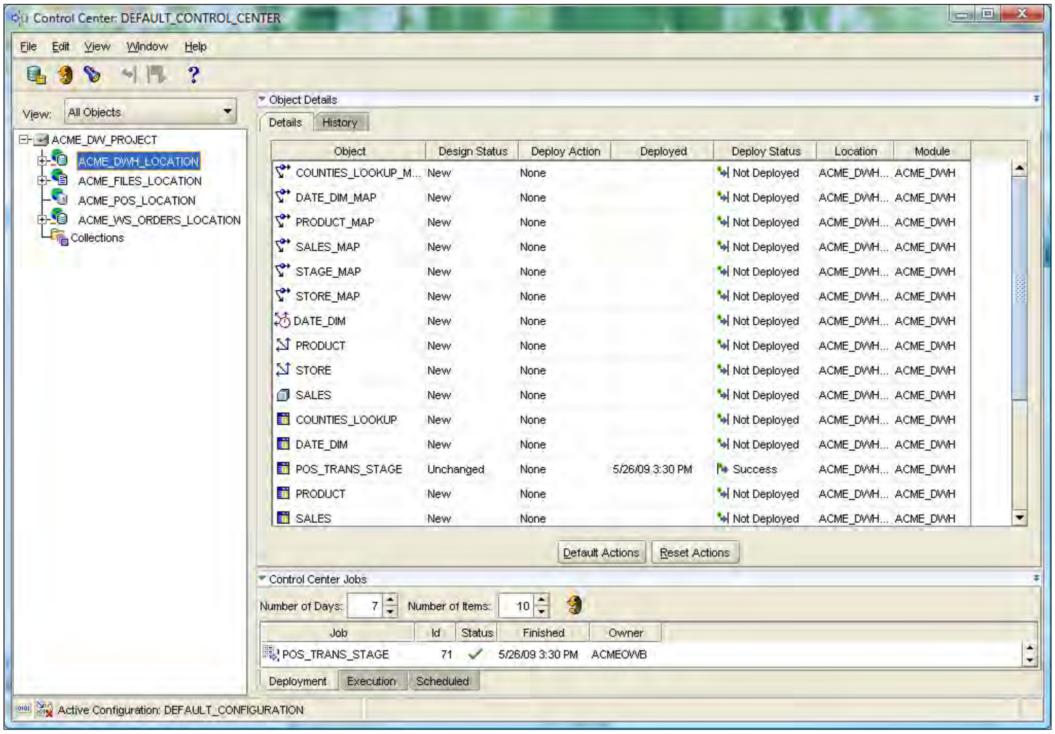
This shows us that the deployment processed successfully with no errors or warnings. But what if the count of errors or warnings was not zero? There would be nothing but a count that would display with this dialog box, so we need a feature to see what these warnings and error messages are. There must be some way to give us more control over the deployment process as the Design Center only shows us design information. This feature of the Warehouse Builder is the **Control Center Manager**.

The Control Center Manager

The Control Center Manager is the interface the Warehouse Builder provides for interacting with the target schema. This is where the deployment of objects and subsequent execution of generated code takes place. The Design Center is for manipulating metadata only on the repository. Deployment and execution take place in the target schema through the Control Center Service. The Control Center Manager is our interface into the process where we can deploy objects and mappings, check on the status of previous deployments, and execute the generated code in the target schema.

We launch the Control Center Manager from the **Tools** menu of the **Design Center** main menu. We click on the very first menu entry, which says **Control Center Manager**. This will open up a new window to run the Control Center Manager, which will look similar to the following:

Download at Boykma.Com


Control Center: DEFAULT_CONTROL_CENTER
File Edit View Window Help
View: All Objects
Object Details
Details History
Default Actions Reset Actions
Control Center Jobs
Deployment Execution Scheduled

ACME_DW_PROJECT
- ACME_DWH_LOCATION
- ACME_FILES_LOCATION
- ACME_POS_LOCATION
- ACME_WS_ORDERS_LOCATION
- Collections

Object	Design Status	Deploy Action	Deployed	Deploy Status	Location	Module
COUNTIES_LOOKUP_M...	New	None		Not Deployed	ACME_DWH...	ACME_DWH
DATE_DIM_MAP	New	None		Not Deployed	ACME_DWH...	ACME_DWH
PRODUCT_MAP	New	None		Not Deployed	ACME_DWH...	ACME_DWH
SALES_MAP	New	None		Not Deployed	ACME_DWH...	ACME_DWH
STAGE_MAP	New	None		Not Deployed	ACME_DWH...	ACME_DWH
STORE_MAP	New	None		Not Deployed	ACME_DWH...	ACME_DWH
DATE_DIM	New	None		Not Deployed	ACME_DWH...	ACME_DWH
PRODUCT	New	None		Not Deployed	ACME_DWH...	ACME_DWH
STORE	New	None		Not Deployed	ACME_DWH...	ACME_DWH
SALES	New	None		Not Deployed	ACME_DWH...	ACME_DWH
COUNTIES_LOOKUP	New	None		Not Deployed	ACME_DWH...	ACME_DWH
DATE_DIM	New	None		Not Deployed	ACME_DWH...	ACME_DWH
POS_TRANS_STAGE	Unchanged	None	5/26/09 3:30 PM	Success	ACME_DWH...	ACME_DWH
PRODUCT	New	None		Not Deployed	ACME_DWH...	ACME_DWH
SALES	New	None		Not Deployed	ACME_DWH...	ACME_DWH

Number of Days: 7 Number of Items: 10

Job	Id	Status	Finished	Owner
POS_TRANS_STAGE	71	✓	5/26/09 3:30 PM	ACMEOWB

Active Configuration: DEFAULT_CONFIGURATION

The Control Center Manager window overview

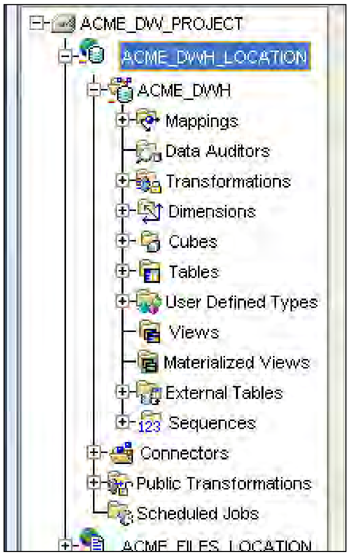
The Control Center Manager interface is organized in a similar manner to the Design Center with multiple windows appearing in the main window. But only two windows are available: the **Object Details** window and the **Control Center Jobs** window. The subwindow on the left that displays the tree hierarchy for our project and the locations defined within it is a permanent part of the interface, and so it does not have a separate window title like the other two.

Beginning with the left subwindow, we see our project name displayed there with a list of the locations that have been defined within our project. The primary location of concern for deployment and execution will be `ACME_DWH_LOCATION`. This is the location we have defined for our target database and selected as default.

Download at Boykma.Com

This is where we can control the registering and un-registering of a location. If we right-click on a location, we see a **Register...** menu entry. When it is selected, it pops up the same dialog box as we have seen previously. If the location has not been previously registered it's connection details will all be editable. But if the location was previously registered successfully, only the password can be set. The **Unregister...** menu entry will remove the connection details for the location.

All the objects defined for this location can be found by clicking on the plus sign to the left of the location name to expand the tree, and then clicking on the plus sign next to the **ACME_DWH** module to display for us a hierarchy of the object types that can be deployed. An example is shown next:



Clicking on the plus sign beside any of the subcategories, such as **Mappings** or **Tables**, will show us the list of the objects of that type defined within our project. If we click on an entry in the hierarchy, the Object Details window will update to display the associated objects. In the previous image of the entire **Control Center Manager** window, we can see that the **Object Details** window contains the entire set of objects defined in our project for the main location because that is what is selected in the tree view on the left. Now as we click on the subcategories, the Object Details window updates to display just the objects within that subcategory. If we further expand the tree view on the left to view the objects in a subcategory, we can click on an individual item and the Object Details window will display the details for just that item.

Download at Boykma.Com

[If this looks familiar, it is because the module view for the ACME_DWH module in the tree is very similar to the view we get in the Design Center when viewing that module. All the objects represented here can be deployed and correspond exactly to the same item in the Design Center.]

The Object Details window

Let's click on the ACME_DWH_LOCATION again in the left window and look at the complete list of objects for our project. The statuses will vary depending on whether we've done any deployments or not, when we did them, and whether there are any warnings or failures due to errors that occurred. If you're following along exactly with the book, the only deployment we've done so far is the POS_TRANS_STAGE table and the previous image of the complete Control Center manager interface shows it as the only one that has been deployed successfully. The remainder all have a deploy status of **Not Deployed**.

The columns displayed in the Object Details window are as follows:

- **Object**: The name of the object
- **Design Status**: The status of the design of the object in relation to whether it has been deployed yet or not
    - **New**: The object has been created in the Design Center, but has not been deployed yet
    - **Unchanged**: The Object has been created in the Design Center and deployed previously, and has not been changed since its last deployment
    - **Changed**: The Object has been created and deployed, and has subsequently undergone changes in the Design Center since its last deployment
- **Deploy Action**: What action will be taken upon the next deployment of this object in the Control Center Manager
    - **Create**: Create the object; if an object with the same name already exists, this can generate an error upon deployment
    - **Upgrade**: Upgrade the object in place, preserving data
    - **Drop**: Delete the object
    - **Replace**: Delete and recreate the object; this option does not preserve data
- **Deployed**: Date and time of the last deployment

Download at Boykma.Com

- **Deploy Status**: Results of the last deployment
    - **Not Deployed**: The object has not been deployed yet
    - **Success**: The last deployment was successful, without any errors or warnings
    - **Warning**: The last deployment had warnings
    - **Failed**: The last deployment failed due to errors
- **Location**: The location defined for the object, which is where it will be deployed
- **Module**: The module where the object is defined

Some of the previous columns will allow us to perform an action associated with the column by double-clicking or single-clicking in the column. The following is a list of the columns that have actions available, and how to access them:

- **Object**: Double-click on the object name to launch the appropriate editor on the object.
- **Deploy Action**: Click on the deploy action to change the deploy action for the next deployment of the object via a drop-down menu. The list of available actions that can be taken will be displayed. Not all the previously listed actions are available for every object. For instance, upgrade is not available for some objects and will not be an option for a mapping.

[The deploy action is what determines the scripts that get generated for an object. The **Create** option will generate only a script to create the object. The **Replace** option, in addition to generating the create script, will cause a drop script to be generated. The **Drop** option will cause only a drop script to be generated. The **Upgrade** option, if available, will generate neither a drop nor a create option but will generate a script with the appropriate upgrade SQL code.]

Download at Boykma.Com

An important note about the upgrade option is that the target user must have certain extra privileges granted to him or her in the database to be able to perform an upgrade. If these privileges haven't been granted, an error will occur when trying to do a deployment with an Upgrade option:

RPE-02257: The following Oracle Roles have not been Granted to the Target User: 'SELECT_CATALOG_ROLE'

RPE-02258: The following Oracle Privileges have not been Granted to the Target User: 'EXECUTE ANY PROCEDURE' 'EXECUTE ANY TYPE' 'SELECT ANY TABLE' 'SELECT ANY DICTIONARY'

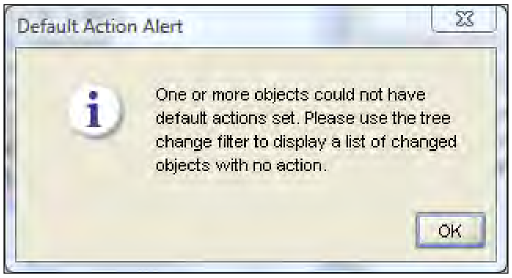
RPE-02259: Please run script <OWB-HOME>/owb/rtp/sql/grant_upgrade_privileges.sql

Fortunately, the Warehouse Builder provides us the previously mentioned script that we can run to grant the appropriate privileges to our target user in the database so that the **Upgrade** option will work. Simply run the script as the system user to grant the privileges.

There are two buttons available in the Object Details window, **Default Actions** and **Reset Actions**. Every object created in the Design Center has a default deployment action associated with it, which is determined by the current design and deployment status. For example, a mapping that has not been deployed yet has a default status of **Create**. A table that was previously deployed but just changed will have a default action of **Upgrade**. The **Default Actions** button will change the displayed **Deploy Action** to show the default action for that object based on a comparison of its design status with its deploy status.

Now let's click on the **Default Actions** button and we'll notice that all the actions are updated to their default action. In our case, with just the `POS_TRANS_STAGE` table deployed and all others not deployed, all the objects except for `POS_TRANS_STAGE` have the **Deploy Action** changed to **Create** from **None**.

After a project has been going on for quite some time and various objects are in various states of deployment and design, it's possible that we might encounter some objects that could not be updated to their default action for whatever reason. The Control Center Manager will notify us of that with the following dialog box:


Default Action Alert

One or more objects could not have default actions set. Please use the tree change filter to display a list of changed objects with no action.

Download at Boykma.Com

This is a rather confusing message. What is the **tree change filter** it is referring to? In the tree window on the left of the Control Center Manager, there is a drop-down menu at the top of the window that is labeled **View**, which is the filter to which it's referring. It defaults to **All Objects** for displaying every object without filtering out any. If we click the drop-down menu and select **Changed Objects**, the objects displayed will update to display only those objects that have been changed or are new.

The previous dialog box is further misleading in that there is no option to list changed objects with no action. There is no filter on the deploy action. We have to list changed objects and then look down through the **Object Details** window for any objects displaying none for the **Deploy Action**. To facilitate this search, we can click on the **Deploy Action** header in the **Object Details** window's column and it will sort the display by the deploy action. It will group all the deploy actions of **None** together and make it easier to find them when scrolling.

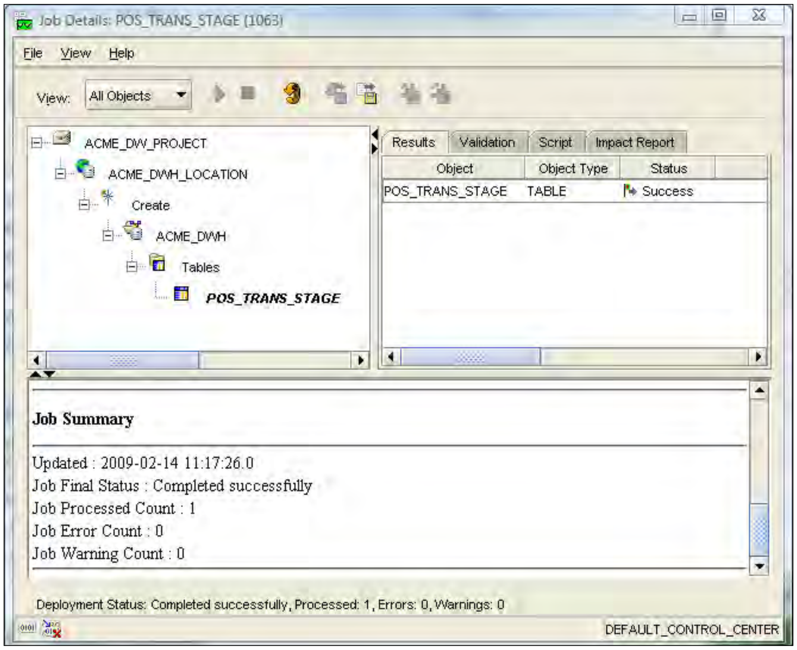
We can then manually specify a deploy action for any object that doesn't have a default action specified if we need to deploy it again.

The other window in the Control Center Manager is the **Control Center Jobs** window. This is where we can monitor the status of any deployments and executions we've performed.

The Control Center Jobs window

Every time we do a deployment or execute a mapping, a **job** is created by the Control Center to perform the action. The job is run in the background while we can continue working on other things, and the status of the job is displayed in the Control Center Jobs window. Looking back at the previous image of the Control Center Manager, we can see the status of the `POS_TRANS_STAGE` table deployment that we performed. The green check mark indicates it was successful. If we want to see more details, especially if there were warnings or errors, we can double-click on the line in the **Control Center Jobs** window and it will pop up a dialog box displaying the details. An example of the dialog box is shown next:

Download at Boykma.Com



Job Details: POS_TRANS_STAGE (1063)

File View Help

View: All Objects

- ACME_DW_PROJECT
  - ACME_DWH_LOCATION
    - Create
      - ACME_DWH
        - Tables
          - *POS_TRANS_STAGE*

Results | Validation | Script | Impact Report

Object	Object Type	Status
POS_TRANS_STAGE	TABLE	Success

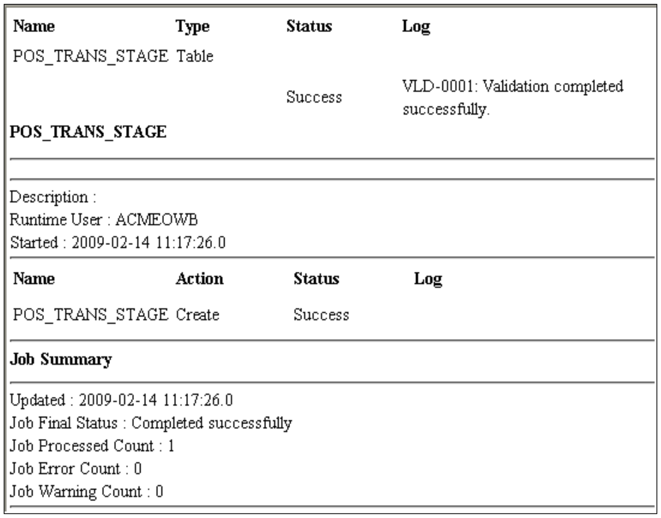
Job Summary

Updated : 2009-02-14 11:17:26.0
Job Final Status : Completed successfully
Job Processed Count : 1
Job Error Count : 0
Job Warning Count : 0

Deployment Status: Completed successfully, Processed: 1, Errors: 0, Warnings: 0

DEFAULT_CONTROL_CENTER

This dialog box is very similar to the previous dialog boxes we looked at for the results we got when validating and generating in the Design Center. The difference is the addition of an extra window at the bottom, which contains the messages that provide the details about the process. It is a scrollable window, and the previous image only shows the last part of the messages. The complete contents of the window are displayed next:



Name	Type	Status	Log
POS_TRANS_STAGE	Table		
		Success	VLD-0001: Validation completed successfully.

POS_TRANS_STAGE

Description :
Runtime User : ACMEOWB
Started : 2009-02-14 11:17:26.0

Name	Action	Status	Log
POS_TRANS_STAGE	Create	Success	

Job Summary

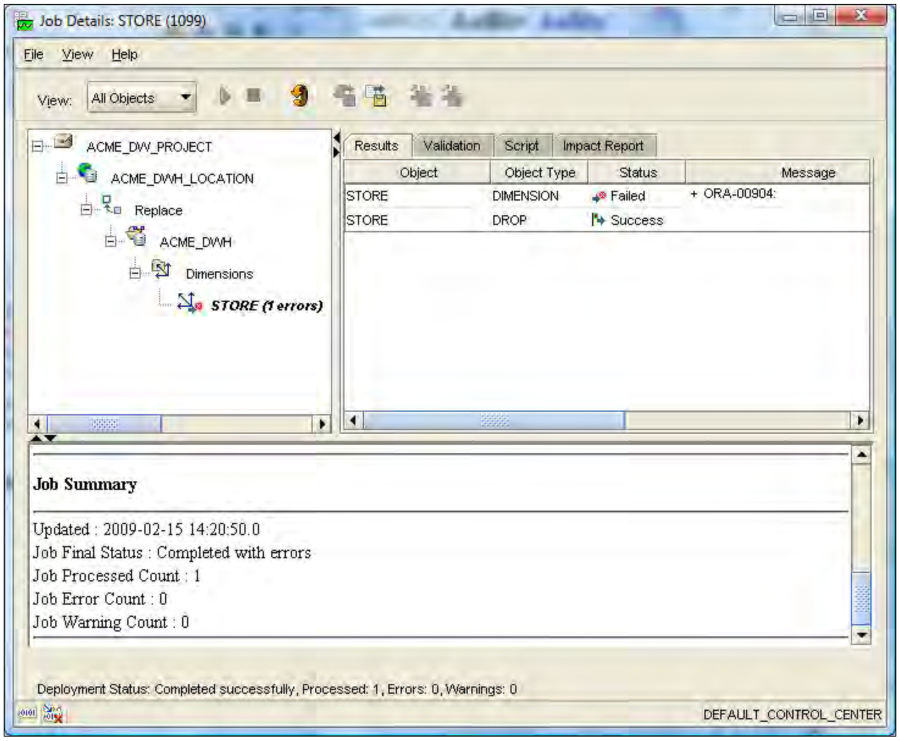
Updated : 2009-02-14 11:17:26.0
Job Final Status : Completed successfully
Job Processed Count : 1
Job Error Count : 0
Job Warning Count : 0

Download at Boykma.Com

We can see from the previous image that the validation messages are also included because when the Warehouse Builder deploys an object, it automatically does actual validation and generation first. It is not necessary to perform a manual validation and then a generation before doing a deployment if we need to redeploy an object, as it will do that for us automatically.

If there were any errors in deployment, these errors would appear in the previous window also. For deployments, as we're dealing with actual scripts executing in the target database to create objects or mapping code, the deployment errors will be of the form of errors generated by the database. These will have an error code of the `ORA-XXXXX` format with `ORA` being a generic indicator that it is an Oracle database error, and `XXXXX` being the five-digit error number zero-padded on the left. More details about these errors can be found in the *Oracle Database Error Messages 11g Release 1(11.1)* guide, which is available online at `http://download.oracle.com/docs/cd/B28359_01/server.111/b28278/toc.htm`. We can look up the Oracle errors easily in this manual by referring to the error number in the table of contents, which is organized by ranges of error numbers in a sequential order.

The following example will illustrate an error encountered when deploying a dimension, which in this case is the `STORE` dimension:



Object	Object Type	Status	Message
STORE	DIMENSION	Failed	+ ORA-00904:
STORE	DROP	Success	

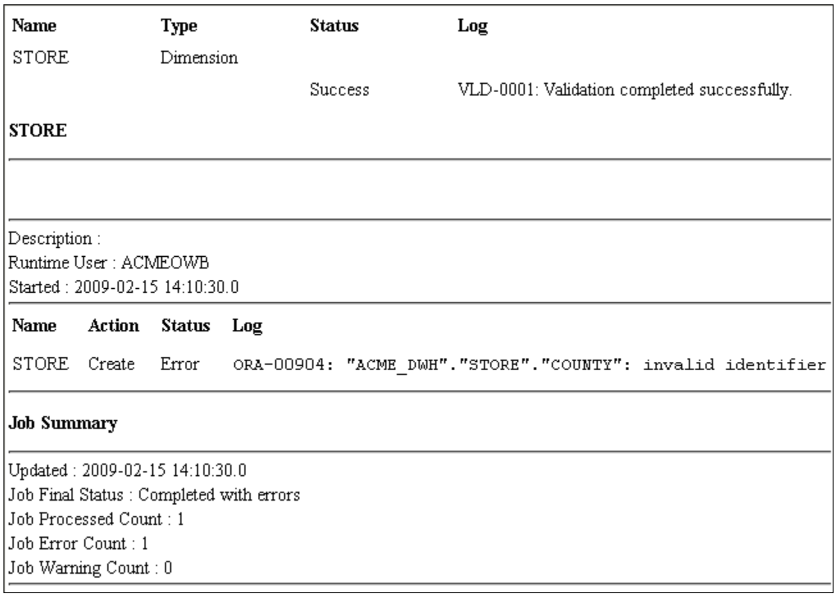
Job Summary

Updated : 2009-02-15 14:20:50.0
Job Final Status : Completed with errors
Job Processed Count : 1
Job Error Count : 0
Job Warning Count : 0

Deployment Status: Completed successfully, Processed: 1, Errors: 0, Warnings: 0

Download at Boykma.Com

This dialog box indicates an error occurred while trying to deploy the `STORE` dimension. There are two results lines because this is a redeployment of an object that had already been deployed before, and Replace was the default deployment action it performed. This involves a `DROP` of the existing object, which was successful, and a `CREATE` of the new version of the object, which failed with an `ORA-00904` error. To see the full error, we can view the contents of the scrolled window at the bottom that displays the full error message:



```
Name          Type          Status        Log
STORE         Dimension
                            Success       VLD-0001: Validation completed successfully.

STORE

Description :
Runtime User : ACMEOWB
Started : 2009-02-15 14:10:30.0

Name    Action  Status  Log
STORE   Create  Error   ORA-00904: "ACME_DWH"."STORE"."COUNTY": invalid identifier

Job Summary

Updated : 2009-02-15 14:10:30.0
Job Final Status : Completed with errors
Job Processed Count : 1
Job Error Count : 1
Job Warning Count : 0
```

This screenshot is telling us that the `COUNTY` identifier used in the dimension is invalid. This particular error was caused by adding the `COUNTY` attribute to the dimension and underlying table, and deploying the dimension with the new `COUNTY` attribute before deploying the underlying table. The dimension definition in the database refers to columns in the underlying table. The metadata in the Design Center was correct—the table definition included the `COUNTY` column—which is why the validation was successful, but the `STORE` table that had been created in the database did not have a `COUNTY` column.

Download at Boykma.Com

Looking up the Oracle errors

Sometimes error numbers don't appear in the most recent edition of the Oracle Error Messages manual. The previous `ORA-00904` error is an example of such an error. Looking in the table of contents in the Oracle Database 11*g* Rel.1 error messages guide referenced previously, the error numbers go up to `ORA-00851` in Chapter 2 and then start with ORA-00910 in Chapter 3. In fact, we have to go all the way back to the *Oracle 9i Database Error Messages Release. 2(9.2)* guide at `http://download.oracle.com/docs/cd/B10501_01/server.920/a96525/toc.htm` to find that error listed.

Some of these errors have been around for quite some time and many new error messages are created with each new database release, so some get left out. In that case, searching older versions of the Error Messages manuals or doing a simple Internet search on your favorite search engine for the `ORA` and `00904` strings (or whatever the error message is) will turn up some additional information.

With this illustration of the Control Center Manager and its windows, we need to discuss how to deploy objects from within the Control Center Manager.

Deploying in the Control Center Manager

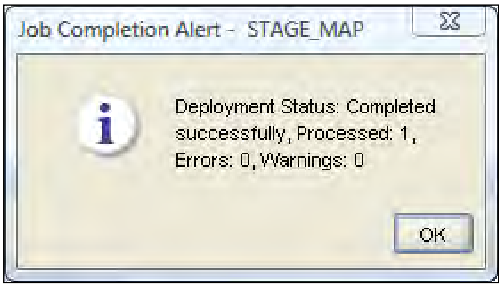
The previous overview of the Control Center Manager windows showed us how it displays the results of our deployments, in particular the ones we initiated from the Design Center, but we can also deploy objects from within the Control Center Manager. This is one of its major functions, along with executing code and checking on the status of jobs.

All of the functions we can perform from the Control Center Manager are initiated from the tree view on the left. There are pop-up menus available on each object and also main menu entries that will perform the action on the currently selected object. Let's deploy the `STAGE_MAP` stage mapping from the Control Center Manager by finding it in the tree view. We have to expand the `ACME_DW_PROJECT` project and the location for our `ACME_DW_LOCATION` target, and then the module for the `ACME_DWH` target database. As we want to deploy a mapping, we need to look under the **Mappings** node. So we expand that entry in the tree view, right-click on it, and select **Deploy** from the pop-up menu. We can also click on it and then select **File | Deploy | To Control Center** from the main menu.

Download at Boykma.Com

The pop-up menu on an object and the main menu in the Control Center Manager will update depending on the deploy action currently set for the objects. If the current deploy action is **None**, the **Deploy** pop-up menu entry and the **File** menu **Deploy** submenus will be grayed out, and will not be selectable. If we wish to deploy in that case, we can change the deploy action using the pop-up menu or change it in the **Object Details** window and the **Deploy** menu entry will become active. We can also just use the **Default Actions** button in the **Object Details** window to set a default deployment action. In this case, it defaults to **Create** as we saw previously and the **Deploy** menu option is now available.

A new entry will be created in the Control Center Jobs window and the status will update as the job progresses. It's possible we might be presented with another dialog box saying a location hasn't been registered yet and it will prompt for the connection information similar to the previous dialog box. The STAGE_MAP references the ACME_POS source SQL Server database using the ACME_POS_LOCATION location, which also needs to be registered. As before, we can just fill in the password for the **ACME_DW_USER** login, double-check the remaining information, and click the **OK** button. Now it will proceed with the deployment. Remember to enclose the password in double quotes as this is a SQL Server database location. When it is completed, we'll be presented with the following pop-up window indicating success or failure if we've configured that option in **Preferences** as previously discussed:


Job Completion Alert - STAGE_MAP

Deployment Status: Completed successfully, Processed: 1, Errors: 0, Warnings: 0

We can close that pop-up window and the status will update to reflect the final result. If we havn't selected the option in preferences to display the completion pop-up window, the status of the job in the Job window simply updates to reflect the success or failure of the job. In either case, we can view details by double-clicking on the job if we need to.

Download at Boykma.Com

Executing

Now we have our staging table deployed to the target database, the `POS_TRANS_STAGE` table, and have successfully deployed the mapping to load that table from our `STAGE_MAP` source database. This means we now have enough of our target database deployed to be able to execute the `STAGE_MAP` mapping to load the staging table. Let's do that now so that we will have progressed through the entire process once. Loading the staging table is the first step we have to take to load our database before we can proceed to load the actual target dimensions and cube. After we execute this mapping, we can go back and deploy the remaining objects, and execute them to load the dimensions and cube.

The process of executing a mapping cannot be performed from the Design Center. To execute mappings, we need to be in the Control Center Manager. However, once in the Control Center Manager, the process of executing is very similar to deploying. Results are displayed in the Control Center Jobs window, which is the same as that of the deployment results, but on a different tab, that is the **Execution** tab.

To execute a mapping we might think to look for a menu entry that says **Execute**, but we will not find it. We need to select the menu entry that says **Start** to start the code running. This menu entry is available from the pop-up menu by right-clicking on an item in the tree view, and from the **File** menu when an item is selected in the tree view.

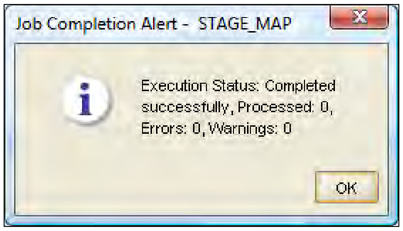
Let's just go ahead and do that now. We'll find the `STAGE_MAP` entry under **Mappings**, which is in the `ACME_DWH_LOCATION` in the tree view. Right-click on it and select **Start**.

When executing code, it's always a good idea to make sure the most recent version of the code has been deployed successfully. Before selecting Start, it is good to just glance at the **Object Details** window for the object, which appears when the object is right-clicked, and make sure that the deployment status shows Success and the design status shows Unchanged.

If we had a problem with the deployment and the status is other than **Success**, we will have issues running it. If the design status shows **Changed**, we don't have the most recent version of the object deployed. We can then fix any issues first, re-deploy, and then execute it.

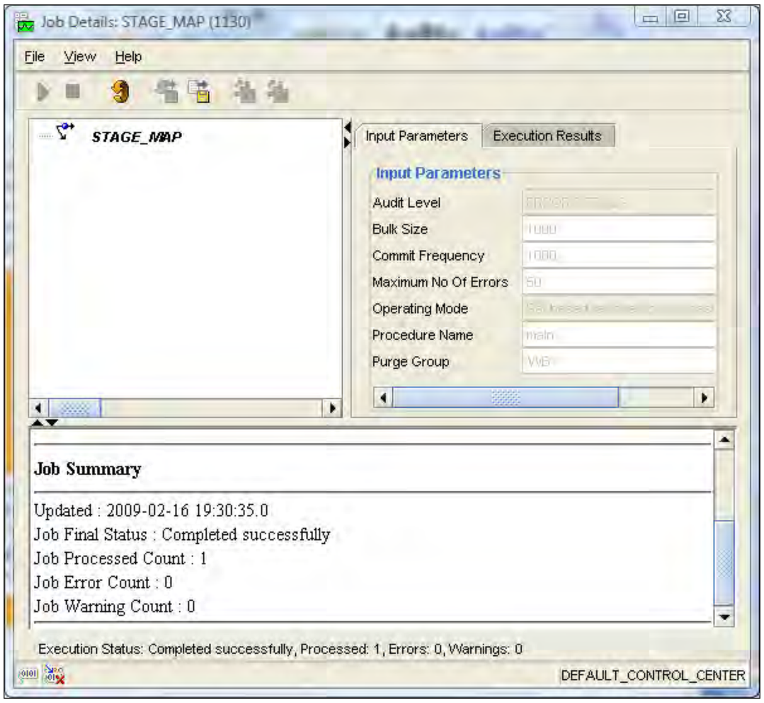
Download at Boykma.Com

Having determined that we have successfully deployed the most recent version, we continue and select **Start**. So the Control Center Manager begins executing the mapping code. As it executes, the first thing we'll notice is that the **Control Center Jobs** window will update to display the **Execution** tab with our newly submitted job as the first entry. Upon completion, if we have checked the preference option to display deployment completion status (which applies to execution status also), we'll get the following results pop-up window:



Execution Status: Completed successfully, Processed: 0, Errors: 0, Warnings: 0

The important thing to notice about this dialog box is the success or failure message. The counts (at least for the processed count) are not accurate. This is a minor bug, as it did indeed process this mapping. This is verifiable by double-clicking on the status for our job in the **Control Center Jobs** window to display the details about this execution. When we do that, we get the following dialog box in which we can clearly see the status and correct counts at the bottom:



Audit Level	[illegible]
Bulk Size	1000
Commit Frequency	1000
Maximum No Of Errors	50
Operating Mode	[illegible]
Procedure Name	main
Purge Group	WB

Job Summary

Updated : 2009-02-16 19:30:35.0
Job Final Status : Completed successfully
Job Processed Count : 1
Job Error Count : 0
Job Warning Count : 0

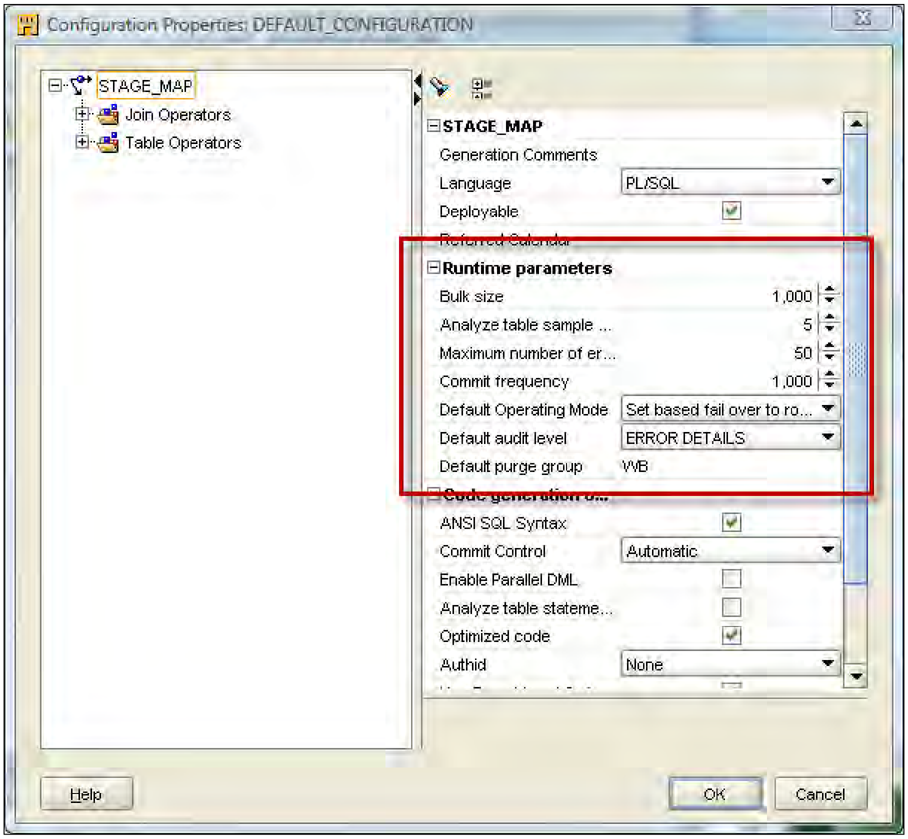
Execution Status: Completed successfully, Processed: 1, Errors: 0, Warnings: 0

DEFAULT_CONTROL_CENTER

Download at Boykma.Com

This is a rather familiar-looking dialog box. We've seen similar ones before with the details of our validations, generations, and deployments. The internal organization of the windows is the same, but the information displayed is customized to the task we're performing. In this case, with the execution of a mapping, the upper left window displays two tabs, one for **Input Parameters** and the other for **Execution Results**.

The input parameters are configuration options for running the mapping that involve the operating mode among others. We discussed the operating mode previously when talking about generating code and viewing the code for the various operating modes. We did not discuss where that operating mode is set as the defaults were adequate for our purpose. The input parameters are accessible by selecting **Configure...** from the pop-up menu by right-clicking on a mapping in either the Design Center or the Control Center Manager. If we do that, we get the following dialog box:

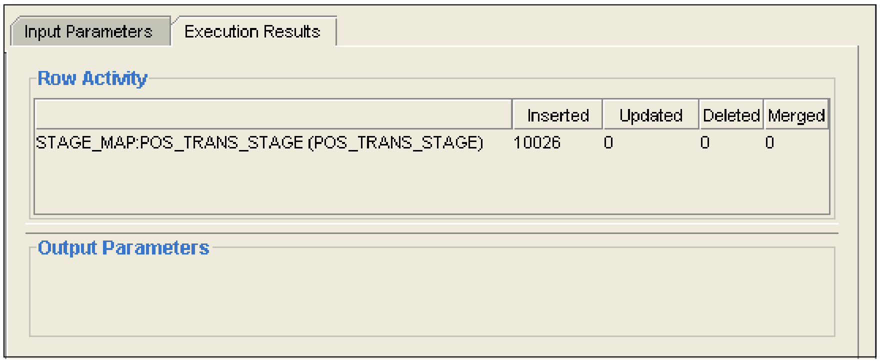


STAGE_MAP
Generation Comments
Language: PL/SQL
Deployable: ✓
Referred Calendar
Runtime parameters
Bulk size: 1,000
Analyze table sample ...: 5
Maximum number of er...: 50
Commit frequency: 1,000
Default Operating Mode: Set based fail over to ro...
Default audit level: ERROR DETAILS
Default purge group: WB
Code generation o...
ANSI SQL Syntax: ✓
Commit Control: Automatic
Enable Parallel DML: ☐
Analyze table stateme...: ☐
Optimized code: ✓
Authid: None

Download at Boykma.Com

The runtime parameters are inside the red box in the previous screenshot. We have covered the default operating mode previously, but the others are all more advanced than we'll have time or the need to cover here. There are good explanations of all the runtime parameters in the online help accessible by pressing the **Help** button. Select the **Configuring Mappings Reference** link and then the **Runtime Parameters** link from the resulting help dialog box to access detailed explanations of all the runtime parameters. For our purpose, the defaults will all be fine.

The next tab in the **Job Details** dialog box for our execution job is the **Execution Results** tab. Clicking on that will show us results similar to the following, which could display more or less for the inserted count depending on how much sample data we actually have in the source database. As of this execution, there were **10026** records in the source `POS_TRANSACTIONS` table:



	Inserted	Updated	Deleted	Merged
STAGE_MAP:POS_TRANS_STAGE (POS_TRANS_STAGE)	10026	0	0	0

The window and column sizes were adjusted so we could see all the information displayed. It provides the name of the mapping and the name of the target that was loaded, and then a count of the number of records processed. In this case, there were **10026** records that were inserted into the table.

Deploying and executing remaining objects

This completes the process of loading our staging table. It's now ready to be used for loading our dimensions and our cube. We've now gone through every process we needed for creating our data warehouse. All that remains is for us to complete the deployment and execution of the remaining objects. The process is the same for all the objects.

Download at Boykma.Com

At this point, the only issue we need to be concerned with is the order in which we deploy and execute the objects. We don't want to deploy and execute a mapping to load a dimension, for example, until we've deployed the dimension itself; otherwise we'll get errors. We can't deploy the dimension successfully until the underlying table has been deployed. We got a small taste of a possible error that can occur due to incorrectly timing our table and dimension deployments earlier in the chapter when we saw the error that could occur when deploying a dimension that had been changed before the modified underlying table was deployed. The dimension specification that was being deployed in that example did not match the actual table in existence in the database, and so an error occurred.

Deployment Order

With that in mind, let's talk about the order in which we should proceed to deploy and execute our objects. The group of objects we have to deploy consists of the following:

- Dimensions
- A cube
- Tables
- Mappings
- Sequences

We want to start with objects that do not rely upon any other objects, and then proceed from there. The only class of objects from the preceding list that doesn't rely upon any others would be sequences, so we'll do them first. Tables are likely the next candidate for deployment, but there could be foreign key dependencies between tables that will cause errors if the tables are deployed in the wrong order; so we need to watch out for that. In fact, the underlying table created for our `SALES` cube has foreign key dependencies upon the three dimension tables and so those must be done before the `SALES` table. The cube will rely upon the dimensions as well as its underlying table, and the dimensions need to have the underlying tables deployed first. So, it looks like the dimensions would be good to do next and then the cube. Finally, the mappings can be done since they depend on the cube, dimensions, and tables. Now that we have figured this out, here's the final list in order of the objects remaining to deploy:

Download at Boykma.Com

- Sequences
    - `DATE_DIM_SEQ`
    - `PRODUCT_SEQ`
    - `STORE_SEQ`
- Tables
    - `COUNTIES_LOOKUP`
    - `DATE_DIM`
    - `PRODUCT`
    - `STORE`
    - `SALES`
- Dimensions
    - `DATE_DIM`
    - `PRODUCT`
    - `STORE`
- Cube
    - `SALES`
- External tables
    - `COUNTIES`
- Mappings
    - `COUNTIES_LOOKUP_MAP`
    - `DATE_DIM_MAP`
    - `PRODUCT_MAP`
    - `STORE_MAP`
    - `SALES_MAP`

We'll go through each of the objects in the order given in the Control Center Manager or the Design Center, and deploy them.

Download at Boykma.Com

Rather than deploying each of the previous objects one at a time, we can make use of the Warehouse Builder's capacity to deploy more than one object at a time. We need to do the previous groupings in order, but within each group the order of the individual deployments is not critical. So, we can click on the node in the Control Center Manager corresponding to the previous groups, and then click on the **Default Actions** button in the **Object Details** window to set the default action. Then we can right-click on the node (**Sequences**, **Tables**, and so on), and select **Deploy** from the pop-up menu to deploy all objects under the node that have a current deployment action set. This will start up a job in the Control Center Jobs window named for the project (**ACME_DW_PROJECT**), which will deploy all the objects under the node. When it completes, we can double-click on the job to display the details for each of the objects if needed.

When complete, we can check the status of everything in the Control Center Manager by clicking on the **ACME_DWH** database module to display all the objects. We can quickly scan down the list to verify that everything got deployed successfully. When that is complete, we'll move on to the next section where we'll execute them.

Execution order

Now that we have all the remaining objects deployed, it's time to execute them to complete our data warehouse project. The execution only pertains to the code that is generated for the mappings. The execution of the code behind all the other objects was done previously when we deployed them to create the objects. For the mappings, the dependency will be determined by the foreign keys that exist in the tables that the mappings are loading. We can't run a mapping without errors to load a table that has foreign key dependencies on other tables before those other tables have been loaded. We know that our `SALES` table has foreign keys to the dimension tables, so we need to run them to load them before doing the `SALES` table. But we also know our `STORE` mapping needs to do a look up of county information from our `COUNTIES_LOOKUP` table, and so that mapping will need to be run before the `STORE` mapping. These are the known dependencies, and armed with this knowledge, we specify our order as follows for executing mappings:

1. `COUNTIES_LOOKUP_MAP`
2. `DATE_DIM_MAP`
3. `PRODUCT_MAP`
4. `STORE_MAP`
5. `SALES_MAP`

Download at Boykma.Com

We'll execute these one by one as the individual order is important. After executing these mappings in the given order, our data warehouse is now complete and ready to be queried.

Summary

That's it! The data warehouse is now complete. We've now completed the work to develop our ACME Toys and Gizmos Company data warehouse. We covered quite a bit of information in this chapter about validating our objects, generating the code for them, deploying to the target environment, and finally executing the code. We were introduced for the first time to the Control Center Manager where we got to interact with the Control Center to deploy and execute objects in our target database environment.

We covered these topics together in this chapter as they are all related. But in actual projects, we will frequently find ourselves performing these steps in an iterative process as we work on the project. We don't have to necessarily wait until the end to perform all these tasks. We can perform some validation and generation as we design each object or mapping.

However, we have one chapter remaining so don't quit yet because we are going to cover some more small details about what we've done so far and add a few minor topics that will help us maximize our use of the tool.

Download at Boykma.Com

Download at Boykma.Com

9
Extra Features

Congratulations on having made it this far and completing the data warehouse implementation! We've now covered all the Warehouse Builder basics that we need to begin building our data warehouses for our organizations. This chapter will deal with some extra topics that can help us get the most out of what we've learned so far and improve our use of the Warehouse Builder. The focus will be on those features that we will find useful as we create more complex data warehouses, and are faced with making changes and updates.

Metadata change management is an important practice we'll want to employ as we make more and more edits and changes to our data warehouse over time, and the Warehouse Builder includes a number of features that can help us with this. We'll look at the **Recycle Bin** for saving deleted objects, copying and pasting objects to make copies for backup or as the basis for new objects, taking snapshots of objects to save the state at a point in time, and the metadata loader facility for making export files that can be saved to a file in a configuration management tool for backup or to transfer metadata.

We'll also take a look at how to keep objects synchronized between the object and its use in a mapping as well as the auto binding of tables to dimensional objects. Lastly, we'll take a quick look at some online references for more information that will help us. Let's begin by talking about some additional aspects of editing mappings and objects in OWB that can help us when we have to make some changes.

Download at Boykma.Com

Additional editing features

We stepped through the process of building our data warehouse from start to finish in this book, but did not address having to go back and make changes to objects or mappings we've already completed. This presents some unique challenges as we saw when we alluded briefly to an error that might occur if changes are made to a dimension, but not the underlying table. Let's talk about the features the Warehouse Builder has that will help us with keeping a track of the various versions of our objects as we make changes.

Metadata change management

Metadata change management includes keeping a track of different versions of an object or mapping as we make changes to it, and comparing objects to see what has changed. It is always a good idea to save a working copy of objects and mappings when they are complete and function correctly. That way, if we need to make modifications later and something goes wrong, or we just want to reproduce a system from an earlier point in time, we have a ready-made copy available for use. We won't have to try to manually back out of any changes we might have made. We would also be able to make comparisons between that saved version of the object and the current version to see what has been changed.

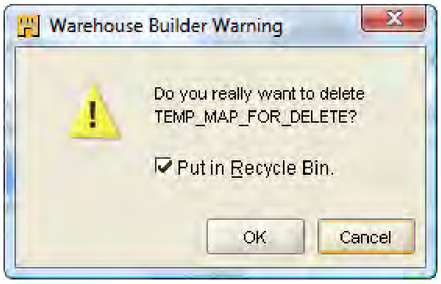
The Warehouse Builder has a feature called the Recycle Bin for storing deleted objects and mappings for a later retrieval. It allows us to make copies of objects by including a clipboard for copying and pasting to and from, which is similar to an operating system clipboard. It also has a feature called **Snapshots**, which allows us to make a copy (or snapshot) of our objects at any point during the cycle of developing our data warehouse that can later be used for comparisons. We'll also discuss making an export file of objects in our project, or even the entire project itself, for backup or copying to another system.

Recycle Bin

The Recycle Bin in OWB is the same concept as that which operating systems use to store deleted files. To try out the Recycle Bin, we need to have an object we can delete. So let's create a temporary mapping object named `TEMP_MAP_FOR_DELETE`. We'll launch the Design Center if it's not already running, and in our `ACME_DWH` module in `ACME_DW_PROJECT` we'll just right-click on the **Mappings** node and select **New**. We'll just close the resulting Mapping Editor that launches for this new mapping because we're just going to delete it next anyway, so it doesn't need to have anything created in it.

Download at Boykma.Com

If we need to remove an object or mapping from our project, there are a number of ways to do that. In the Design Center, we can right-click on an object and select **Delete** from the pop-up menu, or we can click on an object and press the *Delete* key, or we can click on an object and select **Delete** from the **Edit** main menu. Let's perform one of these actions on this new mapping we just created and we'll immediately be presented with the following pop-up screen:



Do you really want to delete TEMP_MAP_FOR_DELETE?

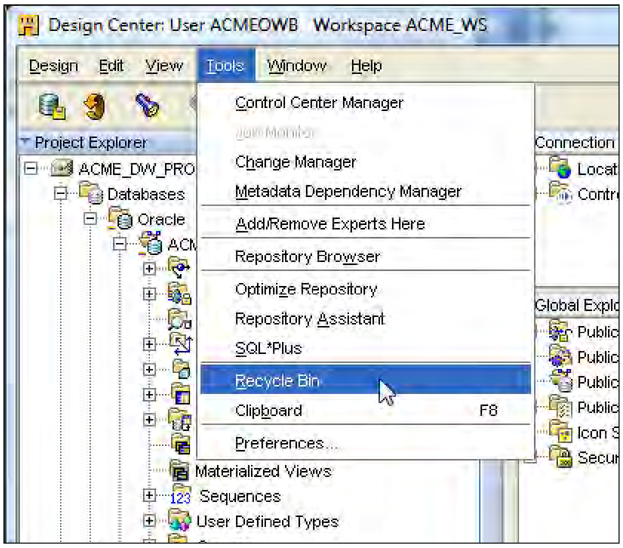
☑ Put in Recycle Bin.

Notice the checkbox for the Recycle Bin. We have the option to delete the object and move it to the recycle bin where it would still be accessible later if needed, or just delete the object entirely so that it is never to be seen nor heard from again. In this case, we created a mapping just to delete, so it really wouldn't matter if it was not put in the recycle bin. But we'll leave the box checked and click on **OK**. We could also click on the **Cancel** button and the dialog box would go away; nothing would happen further. So, leaving the **Put in Recycle Bin** box checked, we'll click on the **OK** button. Now we'll get a quick pop-up window titled **Snapshot Action** that actually mentions taking a snapshot. This is the method it uses to implement the Recycle Bin. That is why we're including both these discussions together here under metadata change management.

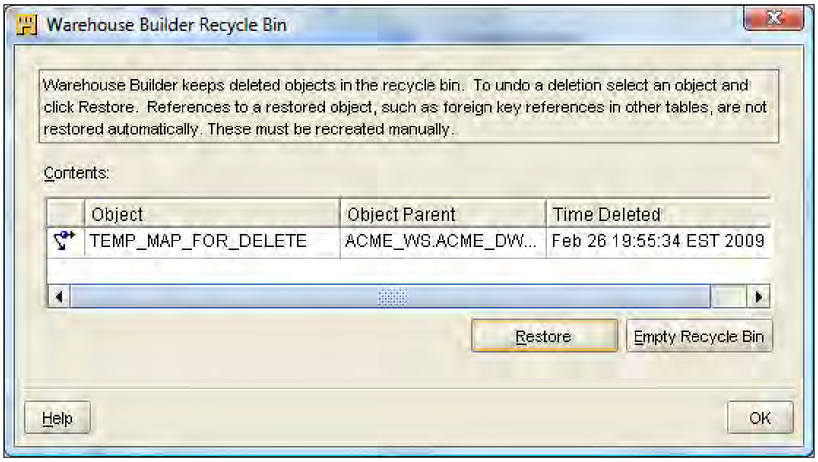
It is important to note that deleting objects in the Design Center, whether placing them in the Recycle Bin or not, does not affect the target database system to which we might have already deployed the object. The delete function in the Design Center affects only the metadata definition stored in the repository workspace of the object, and not the object itself in the target database. To remove that, normal database techniques for removing database objects must be employed.

Download at Boykma.Com

Let's take a look at how to launch the Recycle Bin to recover a deleted object. The **Recycle Bin** is accessible from the main menu of the **Design Center** under the **Tools** menu entry as shown next:



When we click on that menu entry, the Recycle Bin window will pop up as shown in the following screenshot:



Warehouse Builder keeps deleted objects in the recycle bin. To undo a deletion select an object and click Restore. References to a restored object, such as foreign key references in other tables, are not restored automatically. These must be recreated manually.

Object	Object Parent	Time Deleted
TEMP_MAP_FOR_DELETE	ACME_WS.ACME_DW...	Feb 26 19:55:34 EST 2009

Download at Boykma.Com

Let's do that now and take a closer look at it. There may be more objects listed and with different names depending on what has been deleted, but this is what the interface looks like. We can select the object with the left mouse button and click on the **Restore** button to cause that object to be placed back into our project in the same place it was deleted from, as shown by the **Object Parent** column. Of course, the **Time Deleted** is when we deleted the object. It is possible to have more than one entry in the Recycle Bin with the same name. For example, this can happen if we've deleted an object, created a new object using the same name, and then deleted it again. The time it was deleted can clue us into the correct one to restore if needed. The **Empty Recycle Bin** button will do just what it says—clear everything out of the recycle bin. However, this is an all or nothing procedure. We need to make sure we won't ever need anything in the Recycle Bin before clicking on that button because as soon as we click it, we end up with an empty Recycle Bin.

It would be nice if we could selectively remove objects from the Recycle Bin. However, the only way to selectively remove an object is by restoring it and then deleting it again, except this time un-checking the recycle bin checkbox so that it gets deleted permanently.

The Recycle Bin allows us to keep versions of deleted objects. However, if we don't want to delete the object, and instead want to save it before making changes to it, we can make use of the copy and paste feature of the Design Center. Let's close the **Recycle Bin** by clicking on the **OK** button and move on to discuss cut, copy, and paste.

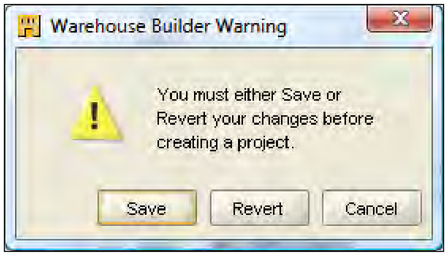
Cut, copy, and paste

We mentioned the Recycle Bin as a concept borrowed from operating systems. Another concept the Warehouse Builder has borrowed from operating systems is the **clipboard**, and **cutting** or **copying** to and **pasting** from the clipboard. Anyone who has used a computer for any length of time has had to recourse to the clipboard where objects and text can be copied between applications. The same concept applies here, but with objects in the Design Center. Even the key combinations that can be used to cut, copy, and paste are the same as for the operating system—*Ctrl+X* to cut, *Ctrl+C* to copy, and *Ctrl+V* to paste.

We can use the cut, copy, and paste features to make a copy of an object in the current project, or to copy an object to another project we might have defined in the Design Center. The only difference between cutting and copying is whether the original object is left in place or not. When cutting, the original object is removed and placed on the clipboard; and if copying, a copy of the object is placed on the clipboard leaving the original intact.

Download at Boykma.Com

Let's walk through a quick example to see this in action. We'll use the option to copy an object between projects by creating a new project just for copying this object there. It's quick and we can remove that project when we are done. To create a new project, we'll click on our `ACME_DW_PROJECT` in the Design Center and then select **Design | New** from the main menu, or click the *Ctrl+N* key combination, or right-click on the project name and select **New**. We may be presented with the following dialog box at this point asking us to either save our work or revert back to the state the project was in when we last saved it. If there have been no changes detected since the last save, we won't see this dialog box:


Warehouse Builder Warning

You must either Save or Revert your changes before creating a project.

We'll click on the **Save** button to save our project and it will immediately present us the dialog box to enter a name for the new project. We'll name our project `ACME_PROJ_FOR_COPYING` and click on the **OK** button to create the new empty project.

We're going to copy a table—our staging table—to this new project, so we need to have a database module defined into which we can copy it. So let's create one now. We'll expand the new project by clicking on the plus sign next to it, and expand **Databases** by clicking on the plus sign next to it.

Notice that our original project, `ACME_DW_PROJECT`, is closed and this new project is opened. We are not able to have more than one project open at a time in the Design Center.

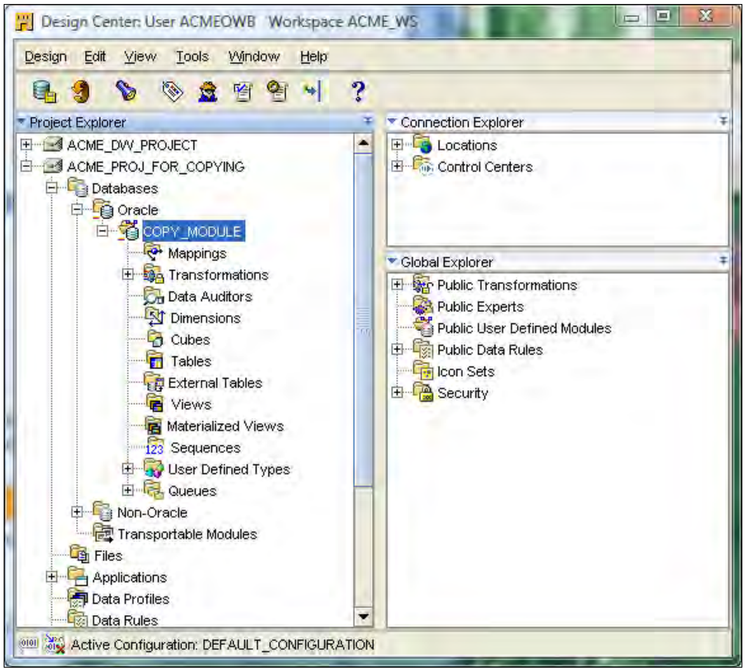
We'll create an Oracle module by right-clicking on **Oracle** and selecting the **New** menu entry to launch the **Create Module Wizard**. We're not going to worry about any specific settings on the following screens and just accept the defaults. This is because we're just using this project to demonstrate some features for this chapter and want to get one created as quickly as possible.

Download at Boykma.Com

However, on the step 1 screen, we do need to give this module a name. It doesn't matter what we call it, so we'll name it `COPY_MODULE`. We'll leave the other options set to their default and click on the **Next** button. The next screen is where we will specify the connection information. Because this is only a temporary module and we're not going to actually have to connect anywhere for real, we can leave the defaults on this screen as they are. The wizard will just create an empty location for us named after the module we just specified with `_LOCATION1` on the end. We'll just click on the **Next** button and then on **Finish** to create the empty database module.

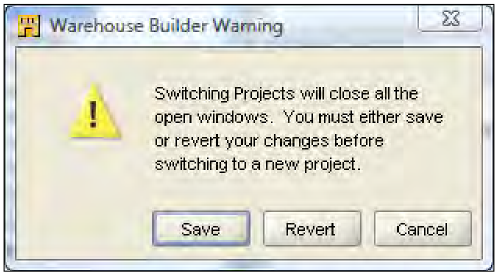
This is a technique for creating a new project that we actually could employ for a real project if we did not have the connection details finalized yet. For example, the actual target database might not be available yet, or maybe we have multiple database servers available and we're not yet sure exactly which one would be used as the target. In that case, we can create a new project quickly just as we did here and leave the connection details unspecified. When the details are finalized, we can edit the connection to fill them in later. We would definitely need to have the connection details specified if we wanted to deploy any objects, but till that point we could create, validate, and generate objects as much as we want.

Let's click on the plus sign next to our newly created database module and we'll see that it has no objects defined in it yet. At this point our Design Center window will look similar to the following:



Download at Boykma.Com

Now that we have a database module, we can copy our table. So let's go back to our real project, `ACME_DW_PROJECT`, that has some objects in it, by clicking on the plus sign next to it. If we have any unsaved changes that we've made to our project, we will see the following dialog box pop up, which we haven't seen yet:



Switching Projects will close all the open windows. You must either save or revert your changes before switching to a new project.

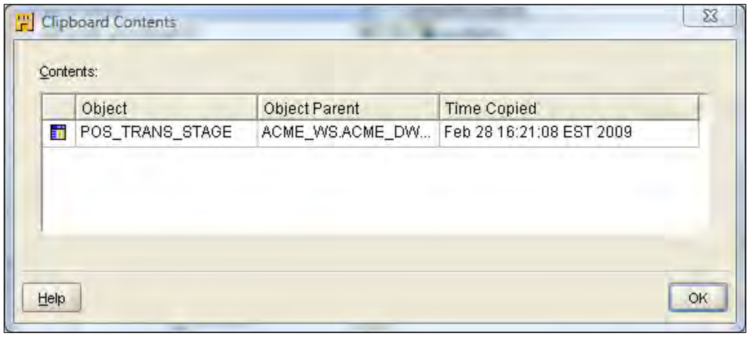
It is similar to the dialog box we just saw when creating a new project and had to save or revert changes. We cannot have more than one project open in the Design Center at the same time, so this is just warning us that it will have to close any open windows we might have for this project, such as an Editor window to edit a mapping or other object, and that we will have to decide whether to save our changes or not. We have the choice to:

- Save our work that we have done so far
- Revert back to the previously saved version of our project
- Cancel the switching of projects and go back to the project

As we have an empty project, we won't have any windows open for it, but we have not saved what we've created so far. We'll need to be sure to click on the **Save** button. That will save the project we just created, close it, and open our original project.

So let's copy the `POS_TRANS_STAGE` table from this project over to our new project that we just created. We'll find it in the **Tables** node in our `ACME_DWH` database module under **Databases | Oracle**. We'll right-click on it and select **Copy** from the pop-up menu, or use one of the other options for copying to copy this table object to the clipboard. We don't want to use the cut option because we don't want to remove the object from our original project. A very quick pop-up window will appear and go away as it's copying. If the window stayed around long enough to read, it would tell us it was copying. Our `POS_TRANS_STAGE` table is now on the clipboard. If we want to verify that, we can select **Clipboard** from the **Tools** main menu or press the *F8* key to pop up the clipboard window to display the contents of the clipboard as shown next:

Download at Boykma.Com

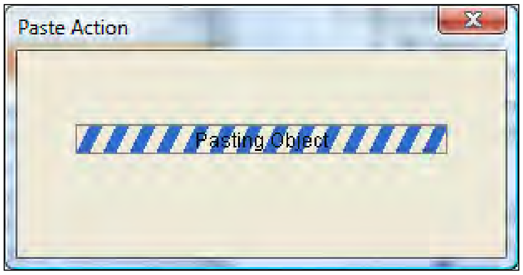

Clipboard Contents

Contents:

Object	Object Parent	Time Copied
POS_TRANS_STAGE	ACME_WS.ACME_DW...	Feb 28 16:21:08 EST 2009

Note that the clipboard can contain only one object at a time. If we cut or copy another object to the clipboard, it will replace this one. So when we're copying multiple objects, we need to be sure to paste from the clipboard before copying another object to it.

Now that we have the `POS_TRANS_STAGE` table on the clipboard, we can paste it into the other project we just created. So let's click on the plus sign beside the new project we created to close our main project and open the new project. The objects that are placed on the clipboard using a cut or copy feature will only be pasted back to the node in the currently open project corresponding to where it was originally cut or copied from. In our case this is a table, so it will be pasted back to the **Tables** node of the new project. Therefore, the **Paste** menu option will only appear on the menu for the **Tables** node. So let's navigate there in the project tree by right-clicking on **Tables** and selecting **Paste**. We can also click on the **Tables** node and type the *Ctrl+V* key combination, or select the **Paste** menu entry on the **Edit** main menu of the Design Center. The Warehouse Builder will now paste the contents of the clipboard into the project, creating a `POS_TRANS_STAGE` table in our new project. It will display the following pop-up window as it is pasting the object:



We could also have pasted the table back into the **Tables** node of our original `ACME_DW_PROJECT` and it would create a copy of the table with `COPY_OF_` prefixed to the table name.

Download at Boykma.Com

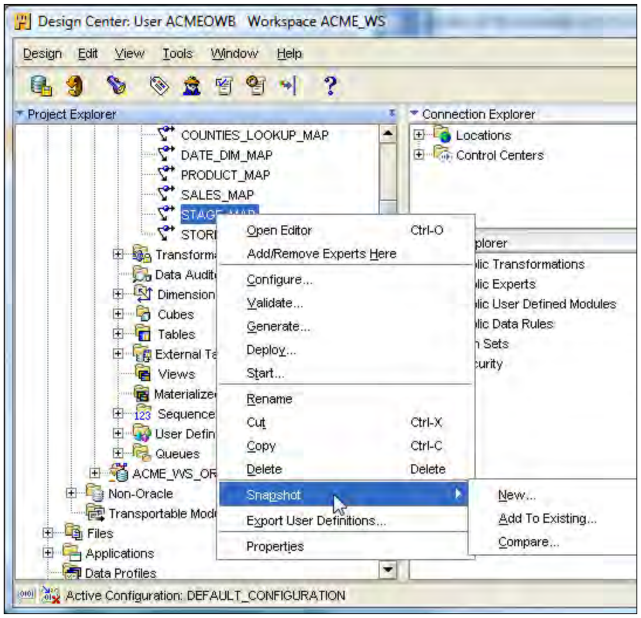
This cut-and-paste technique is very useful for trying out different things with a mapping to see how they will work. If we don't want to risk making edits to our mapping, we can make a copy of it using the copy and paste technique and then edit the copy. It's also possible that we might need two mappings that are almost identical, except for one or two operators. We can develop the mapping, copy and paste it, and then make the edits to the copy to change those one or two operators. We have the whole second mapping created without going back through the entire create process. This can be done with tables and other objects as well and is a very handy time saver.

The contents of the clipboard will not last forever. When we exit the Design Center, the clipboard contents are emptied out. The next time we start Design Center, we start with an empty clipboard. So, we must be very careful not to leave anything on the clipboard that we might need to save. If we want to save an object to have it available for future use without keeping it in our project, we can use the Snapshots feature. Snapshots will also give us the ability to back up our objects and compare them to see what has changed. Let's keep our new project open with the `POS_TRANS_STAGE` table because we're going to make use of it for the next discussion about Snapshots.

Snapshots

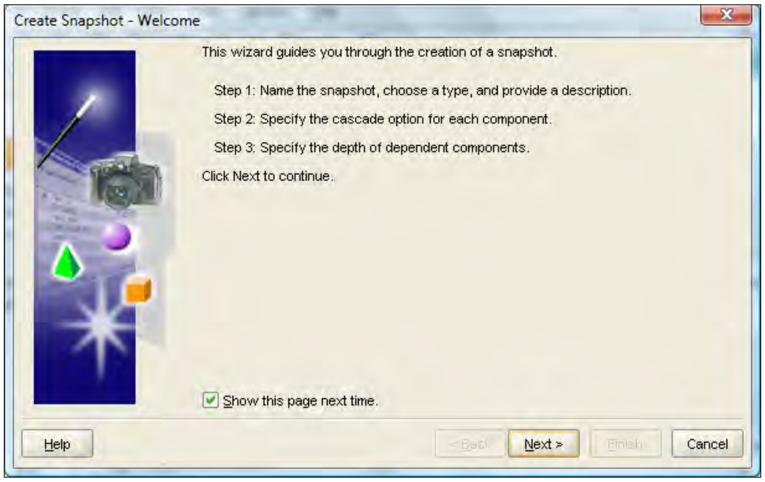
A snapshot captures all the metadata information about an object at the time the snapshot is taken and stores it for later retrieval. It is a way to save a version of an object should we need to go back to a previous version or compare a current version with a previous one. We take a snapshot of an object from the Design Center by right-clicking on the object and selecting the **Snapshot** menu entry. This will give us three options to choose from as shown next:

Download at Boykma.Com



That same menu entry is available on the main menu of the Design Center under the **Design** entry. We can create a new snapshot, add this object to an existing snapshot, or compare this object with an already saved snapshot. The last option is particularly useful for seeing what has changed since the snapshot was taken.

Let's first take a snapshot of our POS_TRANS_STAGE table in the new project we created in the last section. We'll right-click on the table name and select **Snapshot | New...** to create a new snapshot of it. This will launch the Snapshot Wizard to guide us through the three-step process as shown next:


Create Snapshot - Welcome

This wizard guides you through the creation of a snapshot.

Step 1: Name the snapshot, choose a type, and provide a description.

Step 2: Specify the cascade option for each component.

Step 3: Specify the depth of dependent components.

Click Next to continue.

Download at Boykma.Com

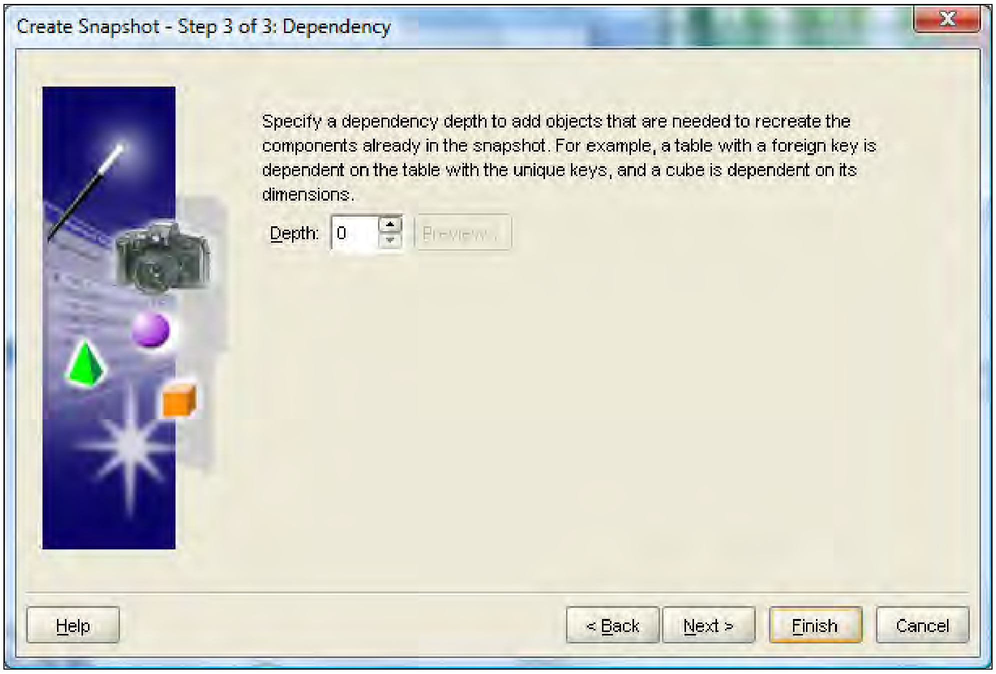
1. We'll click on the **Next** button to move to step 1 of the wizard where we'll give our snapshot a name. This name will be stored in the database as an object name, and so must conform to the Oracle Database requirement that identifiers be no more than 30 characters in length and also must be unique. The wizard will validate the name for us and pop up an error message if we've violated any of the naming conventions, including exceeding the 30-character limit or using invalid characters such as spaces. We'll call our snapshot `POS_TRANS_STAGE_SNAP`. If we like, we can enter a description also to further identify what is in the snapshot.

   There are two types of snapshots we can take: a **full** snapshot that captures all metadata and can be restored completely (suitable for making backups of objects) and a **signature** snapshot that only captures the signature or characteristics of an object just enough to be able to detect changes in an object. The reason for taking the snapshot will generally dictate which snapshot is more appropriate. We can click on the **Help** button on this screen to get a detailed description of the two options. In our case, we'll take a full snapshot this time. Full snapshots can be converted to signature snapshots later if needed, and can also be exported like regular workspace objects. Having selected **Full**, we click on the **Next** button to move to the next step.

2. This step displays a list of the objects we're capturing in this snapshot. We have the option on this screen to select **Cascade**, which applies to folder-type objects. We can take a snapshot of any workspace object, including nodes and even the entire project itself. We can then select **Cascade** to have it include every object contained within that folder object. This is an easy way to capture a large number of objects at once. In our case, we're only capturing the `POS_TRANS_STAGE` table, so **Cascade** would have no effect. We'll click on **Next** and move to step 3, the final step.

3. In the final step we are asked to select a depth to which we'd like to traverse to capture any dependent objects for this object. The screenshot of this step is shown next:

Download at Boykma.Com

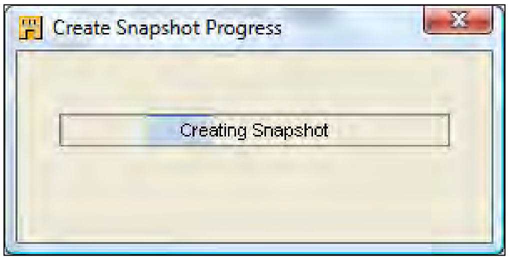

Create Snapshot - Step 3 of 3: Dependency

Specify a dependency depth to add objects that are needed to recreate the components already in the snapshot. For example, a table with a foreign key is dependent on the table with the unique keys, and a cube is dependent on its dimensions.

Depth: 0

Since our POS_TRANS_STAGE table is a standalone table with no foreign key dependencies on any other tables, the default of zero is fine. Even if there were foreign key dependencies, we may not want to capture the additional tables and so would leave it at zero. If we set it to something higher, it will include any object this object depends on. If it is set to something higher than 1, it will proceed to objects that are dependent on those objects, and so on, until it reaches the depth we've specified.

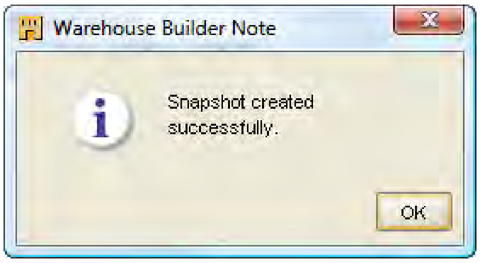
So, leaving it at **0**, we'll click on **Next** and get the summary display showing us what options we chose. Then we'll click on the **Finish** button, which will actually take the snapshot. It will display a progress dialog box showing that it's working, as seen next:


Create Snapshot Progress

Creating Snapshot

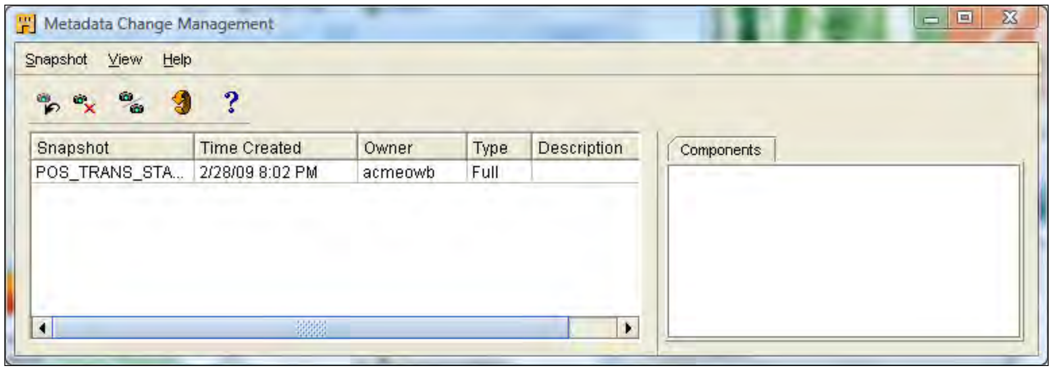
Download at Boykma.Com

When it is done, it will present us with the completion message as shown next:



Snapshot created successfully.

If we want to see what snapshots we've created, there is an interface we can use, which is available on the **Tools** menu of the Design Center. It is called **Change Manager** and will launch the **Metadata Change Management** interface where we can manage our snapshots. It is shown next with our snapshot displayed:



Snapshot	Time Created	Owner	Type	Description
POS_TRANS_STA...	2/28/09 8:02 PM	acmeowb	Full	

If there were more than one snapshot, each would appear in the list on the left. If we click on an entry on the left, the right **Components** window updates to display the objects that are contained within the snapshot. The following can be performed on the snapshots by clicking on them and then selecting the corresponding menu entry under the Snapshots main menu:

- **Restore**: We can restore a snapshot from here, which will copy the snapshot objects back to their place in the project, overwriting any changes that might have been made. It is a way to get back to a previously known good version of an object if, for example, some future change should break it for whatever reason.
- **Delete**: If we do not need a snapshot anymore, we can delete it. However, be careful as there is no recycle bin for deleted snapshots. Once it's deleted, it's gone forever. It will ask us if we are sure before actually deleting it.

Download at Boykma.Com

- **Convert to Signature**: This option will convert a full snapshot to a signature snapshot.
- **Export**: We can export full snapshots like we can export regular workspace objects. This will save the metadata in a file on disk for backup or for importing later.
- **Compare**: This option will let us compare two snapshots to each other to see what the differences are.

Let's try the compare feature. We'll do a comparison between a workspace object in our Design Center project and a snapshot, rather than comparing two snapshots. This will make use of the third **Snapshot** menu entry we saw previously when right-clicking an object in the Design Center and selecting the **Snapshot** menu entry. We'll compare our `POS_TRANS_STAGE` table in our second project with the snapshot we just took. However, let's change something in the table first so that there will be a difference to be found. We could do the comparison now and it would just tell us that the objects are the same. We want to see what it tells us if any differences are found. So let's edit the table in the Data Object Editor by double-clicking on it in the Design Center.

> We don't want to disturb our main project. So we'll make sure the `POS_TRANS_STAGE` table we double-click on is in the new project we just created, and not the main data warehouse project we just built.

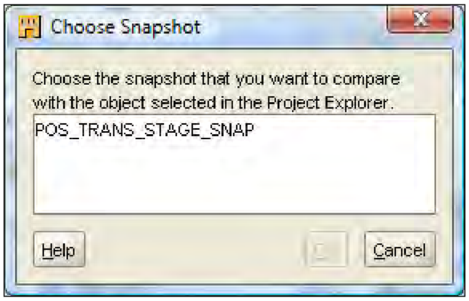
Let's click on the **Columns** tab and scroll down to the end to change the size of the `STORE_COUNTRY` column to 100 from 50, and then close the editor.

> If we leave the editor open, we would have to click on another column or move the focus out of the length field we just changed for the change to actually be detected by the compare function.

When we have made the change and closed the editor, or otherwise moved the focus from the `STORE_COUNTRY` column length field in the editor, we can go back to Design Center. There we can right-click on the `POS_TRANS_STAGE` table and select **Snapshot | Compare...** to compare this object with a snapshot. It will pop up a dialog box listing all the snapshots it found that contain the object we clicked on.

Download at Boykma.Com

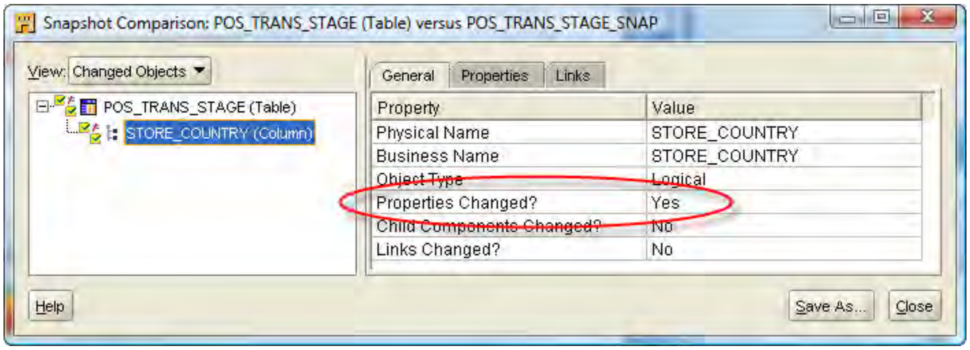
In our case it will display just the one snapshot we created as shown next:



Choose the snapshot that you want to compare with the object selected in the Project Explorer.

POS_TRANS_STAGE_SNAP

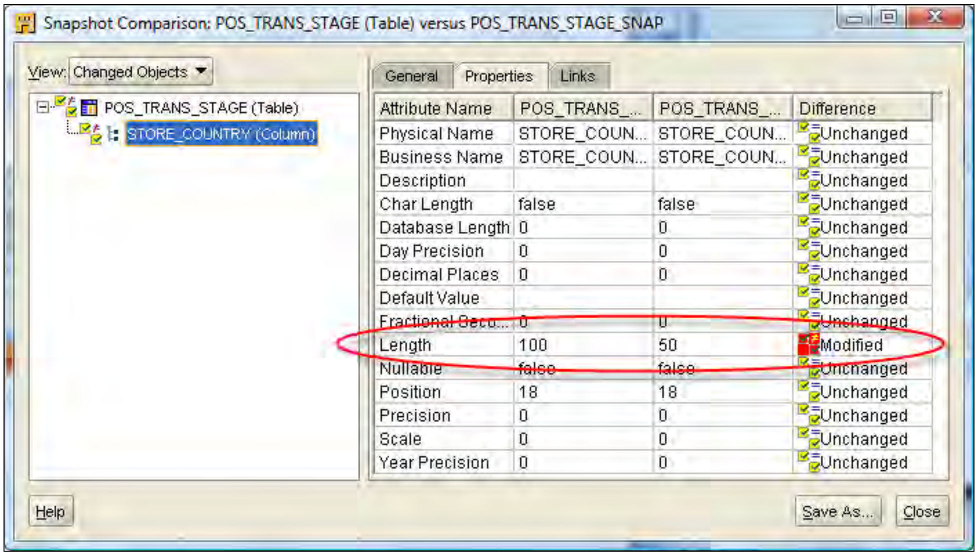
If it did not find any snapshots containing the object we selected, it would tell this to us in a message dialog box. As it found one, we'll click on it to select it and then click on the **OK** button and it will do the comparison, giving us a progress dialog box similar to the previous one we saw when we were creating a snapshot. When the process is complete, it will pop up the results in a snapshot comparison window displaying a tree view on the left of the object with any changed elements in the object shown. On the right, it will display a window with tabs that we can select to view the information about the changes. It shows the `STORE_COUNTRY` column on the left. We'll click on that and the right window will update the tabs with information. The **General** tab will display an overview of the changed element as shown next:



Property	Value
Physical Name	STORE_COUNTRY
Business Name	STORE_COUNTRY
Object Type	Logical
Properties Changed?	Yes
Child Components Changed?	No
Links Changed?	No

It clearly shows that a property (or properties) has (or have) changed. To see the actual change, we can click on the **Properties** tab and it reports that the length of the column has changed from **50** to **100** as shown next:

Download at Boykma.Com



Snapshot Comparison: POS_TRANS_STAGE (Table) versus POS_TRANS_STAGE_SNAP

View: Changed Objects

POS_TRANS_STAGE (Table)
STORE_COUNTRY (Column)

Attribute Name	POS_TRANS_...	POS_TRANS_...	Difference
Physical Name	STORE_COUN...	STORE_COUN...	Unchanged
Business Name	STORE_COUN...	STORE_COUN...	Unchanged
Description			Unchanged
Char Length	false	false	Unchanged
Database Length	0	0	Unchanged
Day Precision	0	0	Unchanged
Decimal Places	0	0	Unchanged
Default Value			Unchanged
Fractional Seco...	0	0	Unchanged
Length	100	50	Modified
Nullable	false	false	Unchanged
Position	18	18	Unchanged
Precision	0	0	Unchanged
Scale	0	0	Unchanged
Year Precision	0	0	Unchanged

The image has been compressed slightly to better fit the page, but it can be expanded on the screen to display the full column headings. They will clearly show `POS_TRANS_STAGE` for the left column with 100 as the length and `POS_TRANS_STAGE_SNAP` for the right column containing the length of 50 characters in the snapshot.

This is a very powerful tool that we can use to manage our metadata changes and stay on top of the changes we've been making. So we'll definitely want to make full use of this feature as we build bigger and more involved data warehouses.

If we want to read information about snapshots, there isn't anything to be found in the User's Guide. Searching for the word **snapshot** will turn up only one quick reference with no explanation. The online help is where we can find more information about it if needed. In the table of contents, look for the section *Reference For Managing Metadata* and expand that. Under that, there is a section named *Managing Metadata Changes*, which is all about snapshots.

Before continuing, we'll make sure to edit the `STORE_COUNTRY` column length to change it back to 50 from 100. We'll then close the **Snapshot Comparison** dialog box by clicking on the **Close** button, and also close the **Metadata Change Management** window if it's still open. Now we'll move on to discuss the import and export of metadata.

Download at Boykma.Com

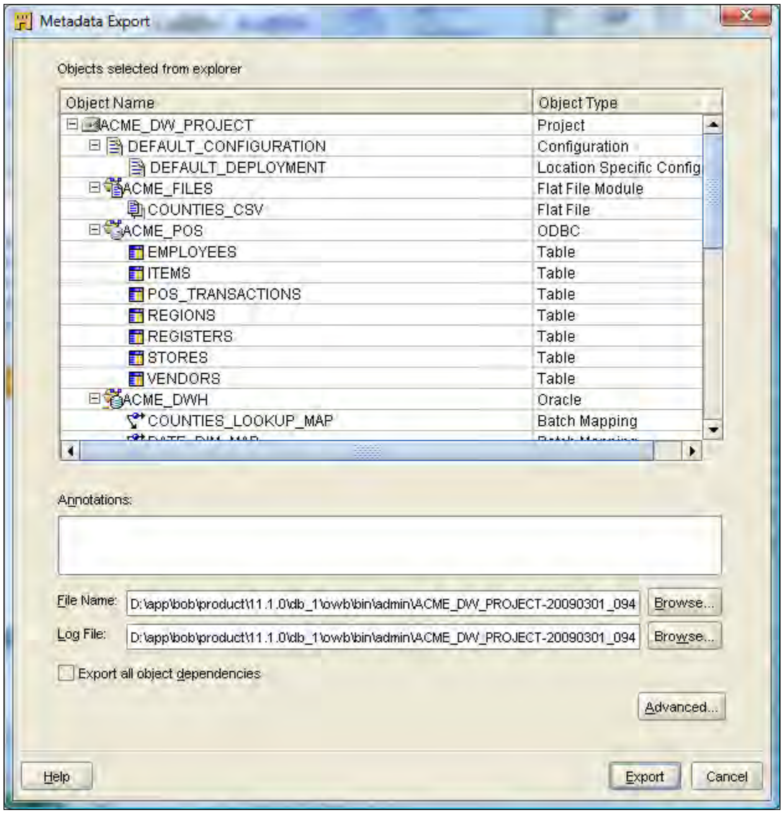
Metadata Loader (MDL) exports and imports

One final change management related tool for managing our metadata that we'll look at in the Warehouse Builder is the ability to export workspace objects and save them to a file using the **Metadata Loader (MDL)** facility. With this feature we can export anything from an entire project down to a single data object or mapping. It will save it to a file that can be saved for backup or used to transport metadata definitions to another repository for loading, even if that repository is on a platform with a different operating system. Some other possible uses for the export and import of metadata are to quickly make copies of workspace objects in multiple workspaces for multiple developers, or to migrate to a newer product version. We would need to export from the old product version, upgrade the Warehouse Builder, and then import back to the new workspace version.

When exporting we can choose any project, node, module, or object in Design Center in either the Project Explorer, Connection Explorer, or Global Explorer windows. If we choose an entire project or a collection such as a node or module, it will export all objects contained within it. If we choose any subset, it will export the context of the objects so that it will remember where to put them on import. That means if we choose a table, for instance, it will include in the metadata the definition for the module in which it resides as well as the project the module is in. We can also choose to export any dependencies on the object being exported if they exist. So an export of tables with foreign key references to other tables, for example, will export these other tables as well to resolve the references.

Let's save an export file of our entire main `ACME_DW_PROJECT` to see how an export is done from the Design Center. We'll select the project by clicking on it and then select **Design | Export | Warehouse Builder Metadata** from the main menu. If we have made any changes, we'll get a dialog box asking us to save the changes or revert the changes as we have seen previously. We'll click on **Save** in that case. The Metadata Export dialog box will be displayed, which will look similar to the following depending on the particular objects that are defined within the project:

Download at Boykma.Com



Metadata Export

Objects selected from explorer

Object Name	Object Type
⊟ ACME_DW_PROJECT	Project
⊟ DEFAULT_CONFIGURATION	Configuration
DEFAULT_DEPLOYMENT	Location Specific Config
⊟ ACME_FILES	Flat File Module
COUNTIES_CSV	Flat File
⊟ ACME_POS	ODBC
EMPLOYEES	Table
ITEMS	Table
POS_TRANSACTIONS	Table
REGIONS	Table
REGISTERS	Table
STORES	Table
VENDORS	Table
⊟ ACME_DWH	Oracle
COUNTIES_LOOKUP_MAP	Batch Mapping

Annotations:

File Name: D:\app\bob\product\11.1.0\db_1\owb\bin\admin\ACME_DW_PROJECT-20090301_094

Log File: D:\app\bob\product\11.1.0\db_1\owb\bin\admin\ACME_DW_PROJECT-20090301_094

☐ Export all object dependencies

Every module, node, and object in our project is depicted in the list for reference, so we can see what will be exported. We also have the opportunity to annotate our export file with some notes. We can use the **Annotations** box to enter any information that we would like to save about the export. It is most often used to save a description of the contents of the export file for quick reference later. Below that we specify the file name of the export file and name for the log file it will create of the export. The dialog box will specify a default file name and location for each, but we are free to change that to any location that suits us.

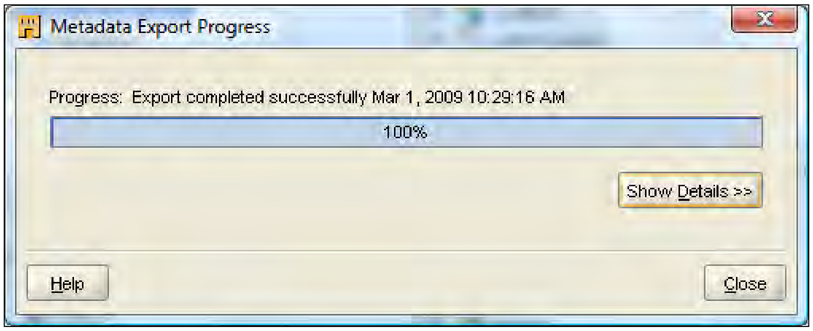
In working with the Warehouse Builder there could be development, test, and production repositories on three different servers. So this feature would be very useful to copy metadata definitions from development to test, and then from test to production. A networked drive and folder that can be accessed from all three servers can be used to store the file, so the MDL file doesn't have to be copied from server to server after saving it.

Download at Boykma.Com

There is also a checkbox on the dialog box for including any object dependencies. As previously mentioned, we can automatically export any objects our selected objects depend on. We might think that all the dependencies are included because we've chosen to export the entire project. But there are dependencies that can exist to objects outside our project such as the locations that are defined in the Connection Explorer or objects defined in the Global Explorer. If we want to automatically include all of them, we can check this box.

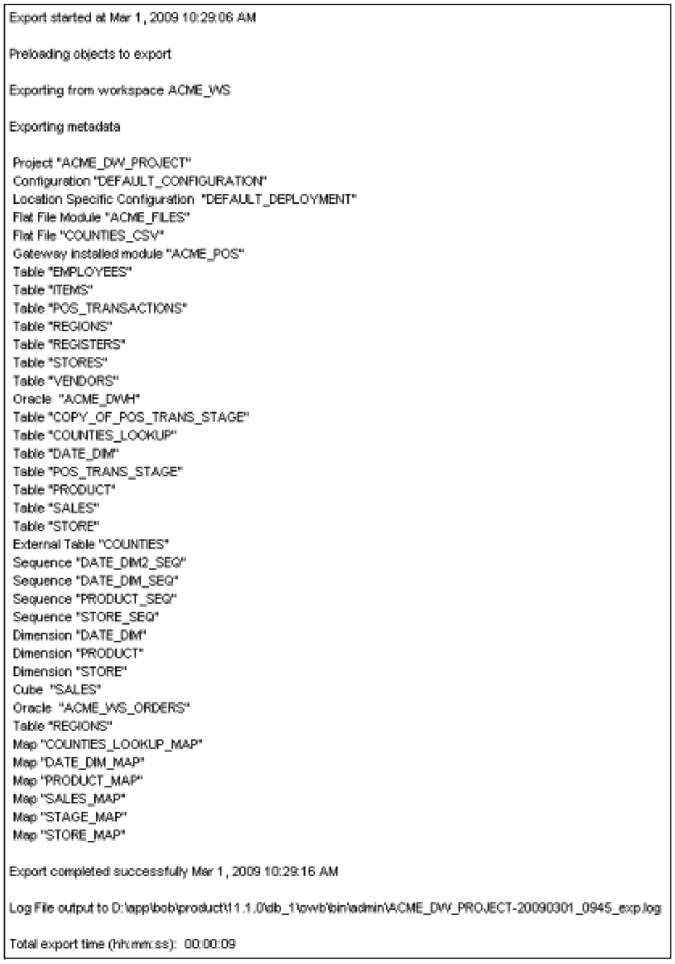
The Warehouse Builder automatically includes an internal project called **PUBLIC_PROJECT** that will contain any of the objects that are external to our individually named projects. These are the objects in the Connection Explorer and Global Explorer that are accessible to any project defined in the workspace.

We'll not export any dependencies so will leave that checkbox unchecked. We'll accept the default file names and locations, and click on the **Export** button. This will start the export and display a progress dialog box with a slider, which will indicate the percent complete. When done, it will look like the following:


Progress: Export completed successfully Mar 1, 2009 10:29:16 AM

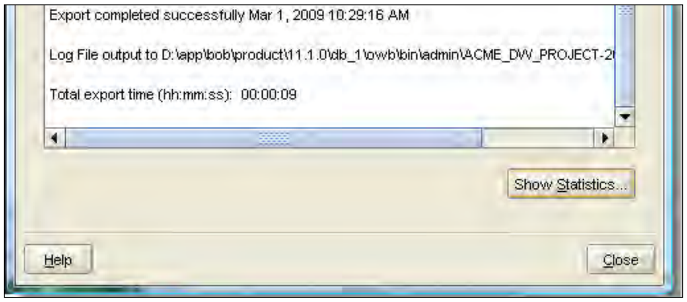
The **Show Details** button will expand the dialog box to display the details of the export in a scrolling window, which shows step by step what it was doing. The following is an example of what that complete dialog box will show:

Download at Boykma.Com



```
Export started at Mar 1, 2009 10:29:06 AM

Preloading objects to export

Exporting from workspace ACME_WS

Exporting metadata

 Project "ACME_DW_PROJECT"
 Configuration "DEFAULT_CONFIGURATION"
 Location Specific Configuration  "DEFAULT_DEPLOYMENT"
 Flat File Module "ACME_FILES"
 Flat File "COUNTIES_CSV"
 Gateway installed module "ACME_POS"
 Table "EMPLOYEES"
 Table "ITEMS"
 Table "POS_TRANSACTIONS"
 Table "REGIONS"
 Table "REGISTERS"
 Table "STORES"
 Table "VENDORS"
 Oracle  "ACME_DWH"
 Table "COPY_OF_POS_TRANS_STAGE"
 Table "COUNTIES_LOOKUP"
 Table "DATE_DIM"
 Table "POS_TRANS_STAGE"
 Table "PRODUCT"
 Table "SALES"
 Table "STORE"
 External Table "COUNTIES"
 Sequence "DATE_DIM2_SEQ"
 Sequence "DATE_DIM_SEQ"
 Sequence "PRODUCT_SEQ"
 Sequence "STORE_SEQ"
 Dimension "DATE_DIM"
 Dimension "PRODUCT"
 Dimension "STORE"
 Cube  "SALES"
 Oracle  "ACME_WS_ORDERS"
 Table "REGIONS"
 Map "COUNTIES_LOOKUP_MAP"
 Map "DATE_DIM_MAP"
 Map "PRODUCT_MAP"
 Map "SALES_MAP"
 Map "STAGE_MAP"
 Map "STORE_MAP"

Export completed successfully Mar 1, 2009 10:29:16 AM

Log File output to D:\app\bob\product\11.1.0\db_1\owb\bin\admin\ACME_DW_PROJECT-20090301_0945_exp.log

Total export time (hh:mm:ss):  00:00:09
```

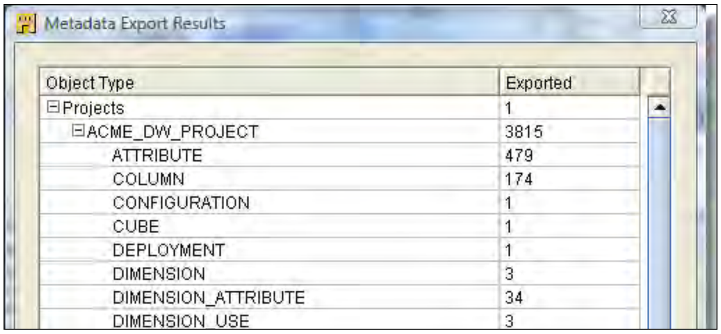
For the statistically minded, we'll find a **Show Statistics...** button now displaying those details as shown next:



```
Export completed successfully Mar 1, 2009 10:29:16 AM

Log File output to D:\app\bob\product\11.1.0\db_1\owb\bin\admin\ACME_DW_PROJECT-2

Total export time (hh:mm:ss):  00:00:09
```

Download at Boykma.Com

If we click on that, we get a dialog box loaded with counts of each individual object and attributes of the objects that were exported. This information is probably more than we'll ever need. But if we focus on the high-level objects such as mappings, tables, dimensions, cubes, and so on, this information can give us a quick validation that we exported what we thought we did—particularly if we've selected a subset of objects and want to make sure we have the correct count. It's just another way of verifying that our export was successful and includes everything we intended. A portion is shown next for reference and we can see that the counts go down to a great level of detail:



Metadata Export Results

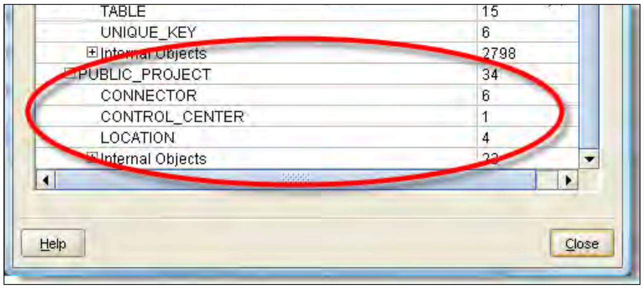
Object Type	Exported
⊟Projects	1
⊟ACME_DW_PROJECT	3815
ATTRIBUTE	479
COLUMN	174
CONFIGURATION	1
CUBE	1
DEPLOYMENT	1
DIMENSION	3
DIMENSION_ATTRIBUTE	34
DIMENSION_USE	3

From this we can see that there is one cube and three dimensions, and this matches with what we created in our project, so that is good. If we're interested in knowing how many columns our tables have in all, or how many individual attributes of objects there are across all the data objects and mappings, or a host of other detailed counts, we have those as well.

The Metadata Loader log file that was created also contains those counts. If we want, we can save them for future reference along with other details about the export.

If we had checked the box for object dependencies on the Export dialog box earlier, we would have seen some additional objects that were exported as shown in the counts here:

Download at Boykma.Com



```
TABLE                15
UNIQUE_KEY           6
⊞Internal Objects    2798
⊟PUBLIC_PROJECT      34
CONNECTOR            6
CONTROL_CENTER       1
LOCATION             4
⊞Internal Objects    23
```

There is that **PUBLIC_PROJECT** we referred to previously. It shows connectors, a control center, and locations that were included. Whether we want these included or not will depend on what we're going to do with the export file. For example, in the previous description about how export files can be used to transfer metadata from the development environment to test and then to production, location details are different for each server. In that case, we wouldn't want to include the location and connector information. As long as we use the same names for our locations and connectors on each server, the project metadata will import with no problems and will function in the same way on all three.

The MDL file that is created is actually in a ZIP format and any application that can open a ZIP file can view the two files contained within it—a file with .mdx extension, which is the file containing the actual objects from the workspace that were exported, and a .xml file that contains the definitions of the exported objects.

That concludes our discussion about managing metadata changes. Now let's discuss an issue that can arise while we're actually making modifications to data objects and mappings, which is the issue of keeping things synchronized among all the objects we've defined.

Synchronizing objects

We created tables, dimensions, and a cube; and new tables were automatically created for each dimension and cube. We then created mappings to map data from tables to tables, dimensions, and a cube. What happens if, let's say for example, a table definition is updated after we've defined it and created a mapping or mappings that include it? What if a dimensional object is changed? In that case, what happens to the underlying table? This is what we are going to discuss in this section.

Download at Boykma.Com

One set of changes that we'll frequently find ourselves making is changes to the data we've defined for our data warehouse. We may get some new requirements that lead us to capture a new data element that we have not captured yet. We'll need to update our staging table to store it and our staging mapping to load it. Our dimension mapping(s) will need to be updated to store the new data element along with the underlying table. We could make manual edits to all the affected objects in our project, but the Warehouse Builder provides us some features to make that easier.

Changes to tables

Let's start the discussion by looking at table updates. If we have a new data element that needs to be captured, it will mean finding out where that data resides in our source system and updating the associated table definition in our module for that source system.

Updating object definitions

There are a couple of ways to update table definitions. Our choice will depend on how the table was defined in the Warehouse Builder in the first place. The two options are:

- It could be a table in a source database system, in which case the table was physically created in the source database and we just imported the table definition into the Warehouse Builder.
- It could be a table we defined in our project in the Warehouse Builder and then deployed to the target database to create it. Our staging table would be an example of this second option.

In the first case, we can re-import the source table using the procedures we used in Chapter 2 for importing source metadata. When re-importing tables, the Warehouse Builder will do a reconciliation process to update the already imported table with any changes it detects in the source table. For the second case, we can manually edit the table definition in our project to reflect the new data element.

For the first case where the table is in a source database system, the action we choose also depends on whether that source table definition is in an Oracle database or a third-party database. If it is in a third-party database, we're going to encounter that same error in the Gateway that we discussed back in Chapter 2 while trying to import source metadata. Hence, we'll be forced to make manual edits to our metadata for that source until that bug is fixed. If the table is in an Oracle database, re-importing the table definition would not be a problem and it will do the reconciliation process, picking up any new data elements or changes to the existing ones.

Download at Boykma.Com

For a hands-on example here, let's turn to our new project that we created earlier while discussing snapshots. We copied our `POS_TRANS_STAGE` table over to this project, so let's use that table as an example of a changing table, as we defined the table structure manually in the Warehouse Builder Design Center and then deployed it to the target database to actually create it. For this example, we won't actually re-deploy it because we'll be using that second project we created. It doesn't have a valid location defined, but we can still edit the table definition and investigate how to reconcile that edit in the next section.

So, let's edit the `POS_TRANS_STAGE` table in the `ACME_PROJ_FOR_COPYING` project in the Design Center by double-clicking on it to launch it in the Data Object Editor. We'll just add a column called `STORE_AREA_SIZE` to the table for storing the size of the store in square feet or square meters. We'll click on the **Columns** tab, scroll it all the way to the end, enter the name of the column, then select **NUMBER** for the data type, and leave the precision and scale to the default (that is 0) for this example.

We can validate and generate the object without having a valid location defined, so we'll do that. The validation and generation should complete successfully; and if we look at the script, we'll see the new column included.

We now need a mapping that uses that table, which we have back in our original project. Let's use the copy and paste technique we used earlier to copy the `STAGE_MAP` mapping over to this new project. We'll open the `ACME_DW_PROJECT` project, answering **Save** to the prompt to save or revert. Then on the `STAGE_MAP` mapping entry, we'll select **Copy** from the pop-up menu. We'll open the **ACME_PROJ_FOR_COPYING** project and then on the **Mappings** node, select **Paste** on the pop-up menu.

We ordinarily won't copy an object and paste it into a whole new project just for making changes. We're only doing it here so that we can make changes without worrying about interfering with a working project.

Synchronizing

Many operators we use in a mapping represent a corresponding workspace object. If the workspace object (for instance, a table) changes, then the operator also needs to change to be kept in sync. The process of synchronization is what accomplishes that, and it has to be invoked by us when changes are made.

Download at Boykma.Com

Now that we have the updated table definition for the POS_TRANS_STAGE table, we have to turn our attention to any mappings that have included a table operator for the changed table because they will have to be synchronized to pick up the change. We saw in Chapter 6 how to create a mapping with a table operator that represents a table in the database. These operators are bound to an actual table using a table definition like we just edited. When the underlying table definition gets updated, we have to synchronize those changes with any mappings that include that table. We now have our STAGE_MAP mapping copied over to our new project. So let's open that in the mapping editor by double-clicking on it and investigate the process of synchronizing.

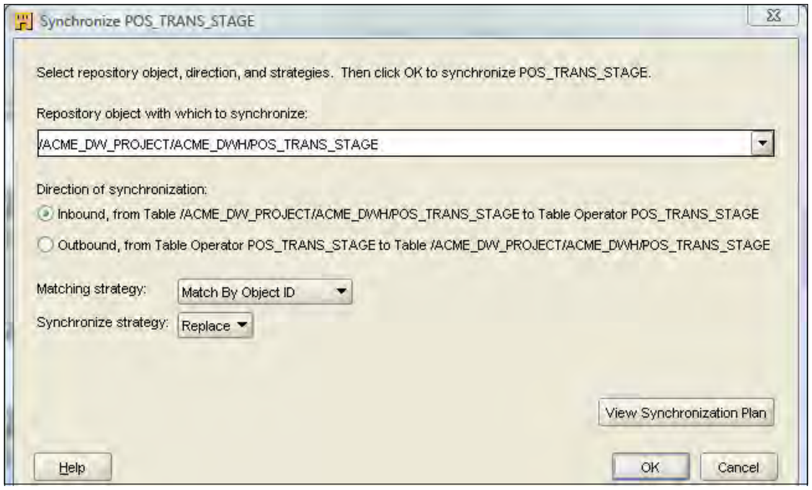
We'll double-check to make sure we've opened the mapping in the correct project as we now have the same mapping name defined in two separate projects. This is perfectly acceptable and any changes we make to one won't affect the other, but we need to make doubly sure that we're in the correct project. In this case we want to be in the ACME_PROJ_FOR_COPYING project, not in the original ACME_DW_PROJECT project. Another reason is that the operators in the mapping still point back to the original object, which we're going to fix by synchronizing; and we don't want to update the wrong mapping.

When we open the mapping for the first time, it may have all the objects overlapping in the middle of the mapping. So we'll just click on the auto-layout button, which we first discussed back in Chapter 5 when looking at the DATE_DIM_MAP mapping. If we look at the POS_TRANS_STAGE mapping operator, we can scroll down the **INOUTGRP1** attribute group or maximize the operator to view all the attributes to see that the new STORE_AREA_SIZE column is not included.

To update the operator in the mapping to include the new column name, we must perform the task of synchronization, which reconciles the two and makes any changes to the operator to reflect the underlying table definition. We could just manually edit the properties for the operator to add the new column name, but that still wouldn't actually synchronize it with the actual table. Doing the synchronization will accomplish both—add the new column name and synchronize with the table. In this particular case there is another reason to synchronize, and that is we copied this mapping into the new project from another mapping where it was already synchronized with tables in that project. This synchronization information is not automatically updated when the mapping is copied.

To synchronize, we right-click on the header of the table operator in the mapping and select **Synchronize...** from the pop-up menu, or click on the table operator header and select **Synchronize...** from the main menu **Edit** entry, or press the *F7* key. This will pop up the Synchronize dialog box as shown next:

Download at Boykma.Com



Synchronize POS_TRANS_STAGE

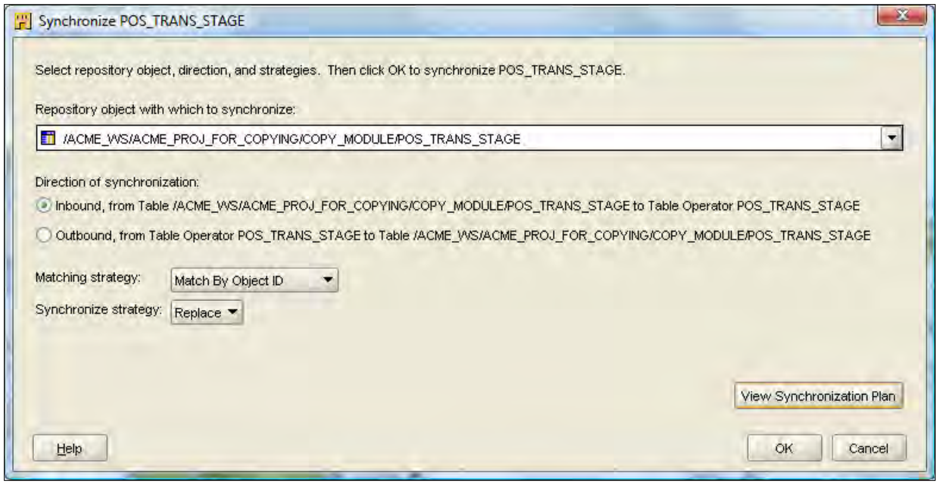
Select repository object, direction, and strategies. Then click OK to synchronize POS_TRANS_STAGE.

Repository object with which to synchronize:
/ACME_DW_PROJECT/ACME_DWH/POS_TRANS_STAGE

Direction of synchronization:
(●) Inbound, from Table /ACME_DW_PROJECT/ACME_DWH/POS_TRANS_STAGE to Table Operator POS_TRANS_STAGE
( ) Outbound, from Table Operator POS_TRANS_STAGE to Table /ACME_DW_PROJECT/ACME_DWH/POS_TRANS_STAGE

Matching strategy: Match By Object ID
Synchronize strategy: Replace

Now we can see why it's so important to make sure we're in the correct project. From the entry indicating the repository object from which it will synchronize, we can see that it's still set to point to the original `POS_TRANS_STAGE` table in the `ACME_DW_PROJECT` project and not the new one we just edited in this project. If we were to rely upon this, we would think we are in the wrong project. We need to click on the drop-down menu and select the `POS_TRANS_STAGE` table in our new `COPY_MODULE`. In fact, this new copy module is the only one we have available. This is good because we wouldn't want to select an object in another module. It's only set that way in this case because it was just copied from that other project. However, we can tell something is a little strange there because the path listed for the `POS_TRANS_STAGE` table stops at `ACME_DW_PROJECT` and no icon is displayed for the type of object. When we select the `POS_TRANS_STAGE` table in our new project, we get the correct display as shown next:



Synchronize POS_TRANS_STAGE

Select repository object, direction, and strategies. Then click OK to synchronize POS_TRANS_STAGE.

Repository object with which to synchronize:
/ACME_WS/ACME_PROJ_FOR_COPYING/COPY_MODULE/POS_TRANS_STAGE

Direction of synchronization:
(●) Inbound, from Table /ACME_WS/ACME_PROJ_FOR_COPYING/COPY_MODULE/POS_TRANS_STAGE to Table Operator POS_TRANS_STAGE
( ) Outbound, from Table Operator POS_TRANS_STAGE to Table /ACME_WS/ACME_PROJ_FOR_COPYING/COPY_MODULE/POS_TRANS_STAGE

Matching strategy: Match By Object ID
Synchronize strategy: Replace

Download at Boykma.Com

This looks much better. Notice how the path includes the workspace name now to fully place it in context. It knows what kind of object it is, a table, so it can display the correct icon. Now we can proceed with deciding whether this will be inbound or outbound.

Inbound or outbound

Now that we have the correct repository object specified to synchronize with, we have to select whether this is an **inbound** or **outbound** synchronization. Inbound is the one we want and is the default. It says to use the specified repository object to update the operator in our mapping for matching. If we were to select outbound, it would update the workspace object with the changes we've made to the operator in the mapping.

Matching and synchronizing strategy

Having decided on inbound, we now have to decide upon a matching strategy to use. The online help goes into good detail about what each of those strategies is, but in our case, we'll want to select **Match By Object Position** or **Match By Object Name**. The **Match by Object ID** option uses the underlying unique ID that is created for each attribute to do the matching with, and that unique ID is not guaranteed to match between projects. It is a uniquely created ID internal to the Warehouse Builder metadata, which uniquely identifies each attribute. The unique ID it stores in the operator for each attribute is the unique ID from the original table it was synchronized with. If we use that option, it will treat all the objects as new because it is not going to get a match on any of them due to using different unique IDs for the copied table.

If we select the **Replace** synchronize strategy, its side effect in the mapping is that all the connections we've made to the existing attributes in the table from the aggregator will be deleted. This is because it has removed all the existing attributes and replaced them with new attributes from the new table with all the new IDs. If we had selected the **Merge** synchronize strategy, it would leave all the existing attributes alone. However, it would add in (or merge in) all the attributes from the new table, in effect duplicating them all in our operator, which is clearly not what we want.

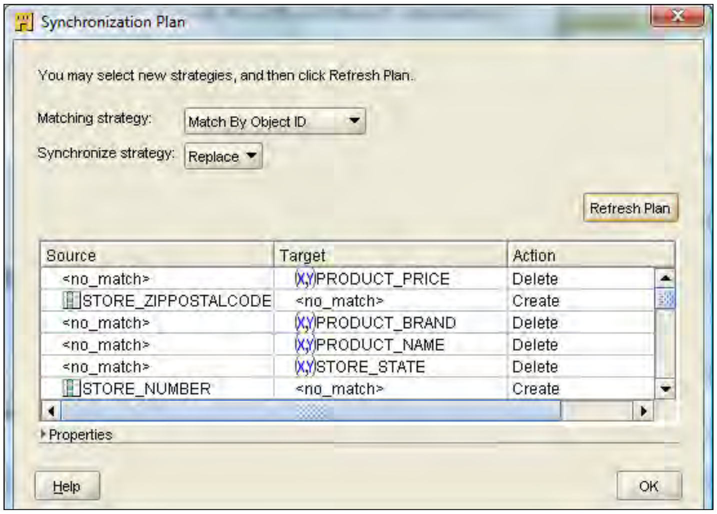
Download at Boykma.Com

Thankfully, there is a solution that will work fine and that is either of the other two **Matching Strategy** selections. By selecting **Match by Object Position**, we'd be telling it to match the operator with the repository object position-by-position, regardless of the unique IDs. So it will not wipe out any connections we've already made as long as there is an attribute in the same corresponding position in the workspace table object. The same holds true for **Match By Object Name**, but this option matches objects by the name of the object and not the position or ID. We know the operator will match all the names and positions of the existing columns, and that the new column has been added to the end. Therefore, we can use either of those two strategies to match and our mapping will remain intact with the existing connections.

With these two options, the **Synchronize Strategy** of merge or replace does not make any difference because all the attributes of the operator in the mapping will be matched in either case. They only indicate what to do with differences. And because the only difference is a new column in the table, regardless of whether we merge in the difference or replace the difference, the net effect is the addition of the new column in the operator.

Viewing the synchronization plan

Based on our selection of the matching and synchronization strategy, the dialog box gives us the option to view what it is going to do before we do it just to be sure we have made the proper selections. We can click on the **View Synchronization Plan** button to launch a dialog box, which will show us what it is going to do. It is nice because we can view the plan without having it actually do anything. So let's select **Match By Object ID** for the matching strategy and **Replace** for synchronization strategy, and click on the **View Synchronization Plan** button. This will launch the **Synchronization Plan** dialog box as shown next:



You may select new strategies, and then click Refresh Plan.

Matching strategy: Match By Object ID

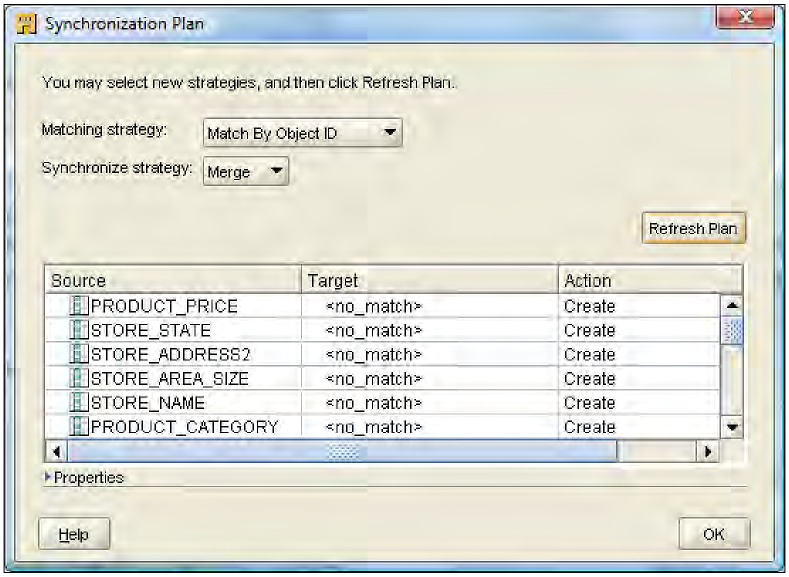
Synchronize strategy: Replace

Source	Target	Action
<no_match>	PRODUCT_PRICE	Delete
STORE_ZIPPOSTALCODE	<no_match>	Create
<no_match>	PRODUCT_BRAND	Delete
<no_match>	PRODUCT_NAME	Delete
<no_match>	STORE_STATE	Delete
STORE_NUMBER	<no_match>	Create

Download at Boykma.Com

The source is the POS_TRANS_STAGE table definition in the workspace and the target is the table operator in the mapping for that table. When matching by object ID, nothing is going to match because the object IDs for the new table are all different from the original table. The Replace option says to replace all differences with the source definitions, so we'll see deletes for all the attributes in the target and creates for all the attributes from the source. That is why all the connections would disappear from our mapping if we used this option.

Let's try the Merge synchronize strategy option with the object ID match by changing that drop-down in the dialog box to **Merge** and clicking on the **Refresh Plan** button. Remember that no actual changes are being made here; it is only telling us what it would do if we made those selections in the main dialog box and clicked on **OK** there. This option will display the following:



You may select new strategies, and then click Refresh Plan.

Matching strategy: Match By Object ID

Synchronize strategy: Merge

Source	Target	Action
PRODUCT_PRICE	<no_match>	Create
STORE_STATE	<no_match>	Create
STORE_ADDRESS2	<no_match>	Create
STORE_AREA_SIZE	<no_match>	Create
STORE_NAME	<no_match>	Create
PRODUCT_CATEGORY	<no_match>	Create

Here we can see that it is creating new entries in the target for all the table entries as it didn't find any matches. It is leaving all the existing target attributes alone, thus merging in the differences. This is clearly not what we want because as indicated previously, it will add in duplicates of every attribute.

If we select either **Match By Object Position** or **Match By Object Name** and refresh the plan, we'll see that it lists one action of **Create** for the new column. There may be updates for existing columns that match, but there should be no other creates or deletes showing. This is what we want, so we'll click on the **OK** button to close the Synchronization Plan dialog box. Back in the main Synchronization dialog box, we'll select **Match By Object Name** as the matching strategy and **Replace** as the synchronization strategy.

Download at Boykma.Com

If this had been an actual update that we had source data for, we would make sure the source data table definition in the workspace was updated to reflect the new data element. As this is related to the store, in our case it would probably have been the STORE source table in the ACME_POS module under **Databases | Non-Oracle | ODBC**. We would then perform the same synchronization operation we just performed on the STORES table operator. We would then have to map that new value through the joiner and to the new column in the POS_TRANS_STAGE table.

We will move along now and discuss one final change and the feature the Warehouse Builder provides for handling it. It is the changing of dimensions and their underlying tables, and keeping them properly bound.

Changes to dimensional objects and auto-binding

When we created our dimensional objects (dimensional in this context being used to refer to both dimensions and cubes) back in Chapter 4, we saw how it automatically created a matching relational table for us. This would be used to hold the data for the object because we had selected ROLAP for the storage implementation. This table was then bound to the dimension with dimension attributes, and levels bound to columns in the table.

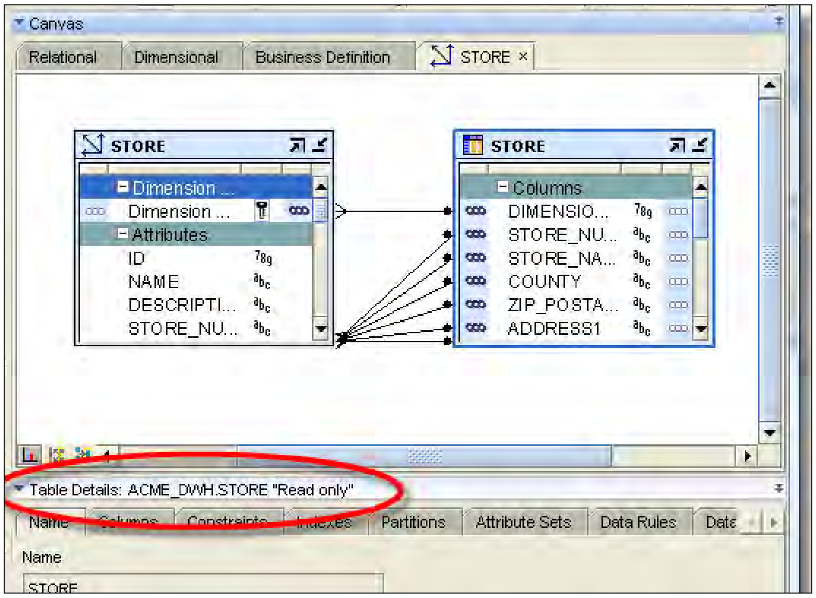
If we carried our previous example one step further, we'd need to add an attribute to our STORE dimension to hold the new value for the size of the store. This would mean we would have to make sure the STORE dimension and the STORE table stayed properly synchronized. This is not quite the same concept as we just discussed. We are now talking about two data objects, and not a data object and an operator in a mapping. That is why the Warehouse Builder generally refers to this as binding instead of synchronizing.

Let's begin by setting up our ACME_PROJ_FOR_COPYING project with copies of the STORE dimension and STORE table for trying this out. We're going to copy the STORE dimension over to the new project and leave the STORE table behind because, as we'll see in a moment, that is going to get automatically generated.

Now that we have the dimension copied over, there is a bit of housekeeping we need to do first. As with the mapping having a reference back to the table in the original project, our STORE dimension will still be bound back to the STORE table in the ACME_DW_PROJECT, and we need to fix that first before continuing with our example. This will be good for us to get more practice working with objects in the Data Object Editor. So let's open the STORE dimension in the editor by double-clicking on it.

Download at Boykma.Com

If we right-click in the header of the dimension in the **Canvas** and select **Detail View...**, another tab is created in the **Canvas** named for the dimension and it now displays both the dimension and its underlying table; however, we never copied over the table. If we click on the table, the window details pane updates to display the information about the table. It is showing us that the dimension is still bound to the original table in our main project we just copied from. This is demonstrated in the following image:



Clearly, that is not good. We need to sever the connection with that table and rebind it to a table in the current project. So let's click on the **Dimensional** tab, right-click on the header of the **STORE** dimension, and select **UnBind** from the pop-up menu. This will switch us right back to the **STORE** tab, but the `STORE` table will no longer be there. Now we need a table to be bound to this dimension. This is where the **Auto Bind** function comes in. With Auto Bind, we can have the Warehouse Builder automatically create the table for us with all the dimension attributes properly bound. To do this, we need to be back on the **Dimensional** tab.

Download at Boykma.Com

There is no officially documented process or function for manually binding a dimension and table together. However, it's possible to do it via a feature called **Experts**, which is one of the more advanced topics we can't cover in this introductory book. There is an Expert that has been specifically developed already to do just that. It can be found on the Oracle Technology web site on the Warehouse Builder Utility Exchange at `http://www.oracle.com/technology/products/warehouse/htdocs/OWBexchange.html`.

In the product area box, click on the **Experts** button and then on the **Search** button. It is the *Create Dimension Expert* utility that has an **Expert** to manually bind a table and dimension. The version says 10*g*R2, but it will still work in 11*g*R1, which we're using for this book.

This is a site Oracle maintains for Warehouse Builder tips, features, code, utilities, and so on that are not found in the official release and are not officially supported. However, much of the content (including the previously referenced Expert) is developed by the Oracle developers themselves who actually work on the Warehouse Builder. We can find a lot of good stuff on that site, so need not fear the fact that Oracle says it's "unsupported".

Let's click on the **Dimensional** tab. Now right-click on the `STORE` dimension and select **Auto Bind** from the pop-up menu. This will create a new `STORE` table for us, automatically bind the existing dimension attributes and levels to columns in the table, and switch us back to the **STORE** tab showing us the details. Now if we click on the `STORE` table, it will show `COPY_MODULE` as the module and not `ACME_DWH`. We can now proceed with our previous example about adding a column. In this case, we'll want to add a column to the `STORE` dimension to save the size value, So let's go back to the **Dimensional** tab to do that.

We could have saved ourselves an Auto Bind step here by just editing the dimension before we did the first Auto Bind. But the intent was to re-create the situation as it would be for real if we had to edit a dimension that was already bound to a table, which is a real-world situation we'll run into quite frequently. The first Auto Bind was just to set up that scenario.

On the **Attributes** tab, scroll down to the end and enter a new attribute called `AREA_SIZE`. Change the data type to **NUMBER** with the precision and scale set to zero. We'll make it an attribute of the `STORE` level. So click on the `STORE` level on the **Levels** tab and scroll down the attributes, and check the box beside the `AREA_SIZE` name.

Download at Boykma.Com

The view of the dimension and table on the **STORE** tab is **"Read-only"**, meaning we can't make any changes there. Switching back to the **Dimensional** tab, we can make changes. However, the Data Object Editor can sometimes still show **"Read-only"** on the **Dimensional** tab. Simply refresh the view by right-clicking on the Canvas window background and selecting **Refresh** to clear the read-only setting. Sometimes it can also help just to click in the canvas outside the dimension and then reselect the dimension to redisplay the attributes below. The preceding changes can now be made on the **Dimensional** tab of the canvas.

Let's save our work and go back to the **STORE** tab to check the `STORE` table, and we'll see that there is no `AREA_SIZE` column. On the **Dimensional** tab, we need to perform the **Auto Bind** again on the dimension, and that will update the table to include the new column. We do not need to do the **UnBind** this time because the correct table is bound; we just want it updated in place.

After the Auto Bind, the table has been updated now to include the new column. We can verify this on the **STORE** tab by inspecting the table.

If this were a working project we had previously deployed, we would need to deploy this updated table and the dimension to actually update the database. We would also need to perform the synchronization (which we discussed in the previous section) in any mappings that included a dimension operator for the `STORE` dimension, so any mapping operators that referenced the dimension would be up to date.

When making a change for real like this, make sure the deployment action for the table is set to **Upgrade** and not **Replace** if there is data in the table already, else the deployment will fail. It should default to **Upgrade**.

However, watch out for the error that may occur when trying to do a deployment with an Upgrade status that we discussed in the last chapter.

This completes our discussion of some additional editing features that we can use as we develop and maintain increasingly complicated data warehouses. We have touched upon just the basics about the Warehouse Builder in this book that we need to know to be able to use it to construct a data warehouse. There is a wealth of more advanced features and capabilities we did not have time to cover in this book, so we just mentioned a few along the way. But this lays the groundwork and has equipped us with the ability to build a complete working data warehouse. There are a lot of resources on the Internet to help us further our education about the tool and to provide assistance if we have questions, much of which Oracle provides directly from its web sites. We'll finish up the book with a brief discussion of some of these resources.

Download at Boykma.Com

Warehouse Builder online resources

Oracle provides a number of resources to assist us with using the Warehouse Builder and with Data Warehousing in general. We saw one such resource earlier in this chapter when we talked about binding a table to a dimension. That download is just one of many that are available from that OWB Utility Exchange tips and tricks web site, which is available directly at the link provided previously or as a link off the much larger web page Oracle has that is devoted entirely to Oracle Warehouse Builder at `http://www.oracle.com/technology/products/warehouse/index.html`.

The link to the download is in a box to the right labeled **Other Downloads** along with some links to software, patches, and sample code.

The OWB developers maintain a blog that they frequently update with various news and notes about OWB—its future or features of interest. The blog is available via a link from the above OWB page or directly at `http://blogs.oracle.com/warehousebuilder/`.

This is an excellent source to get the latest information about OWB directly from the Oracle managers and senior developers who work on the Warehouse Builder. Recently, there is much talk around Oracle and the user community about Oracle's plans relating to their Data Integrator software and the Warehouse Builder software, and how they are moving toward combining the two. The blog has recent posts about how neither tool is going away and how investment in either tool right now is a good investment for years to come.

If we have questions, there is an Oracle Forum devoted specifically to the Warehouse Builder at `http://forums.oracle.com/forums/forum.jspa?forumID=57`.

Oracle developers as well as many from the user community who have been using OWB for a long time frequent the forum and are willing to answer questions.

The Warehouse Builder is just one tool in a large suite of applications that supports the area of business intelligence and data warehousing as a whole. Oracle's web page devoted to that topic, which includes OWB, is at `http://www.oracle.com/technology/tech/bi/index.html`.

If we want more information about the larger topic of business intelligence and data warehousing in general, this is a good place to start.

Download at Boykma.Com

Summary

In this chapter we've finished discussing some additional features of the tool. These are not necessarily essential to the initial development of a data warehouse, but are nevertheless valuable features to have available for further development and maintenance. These include features such as metadata change management, which become critical as more and more changes are required to a data warehouse. The Recycle Bin, the Cut, Copy, and Paste features, Snapshots, and the Metadata Loader all assist greatly in our efforts to control the changes we have to make and to keep a track of prior revisions.

Another valuable feature is the ability to keep the objects synchronized with the operators in the mappings that refer to those objects. Also, we can automatically update objects that are bound together, such as dimensions and the tables used to implement them. These features will assist greatly in the task of making changes to our data warehouse, which will inevitably need to be done in any data warehouse project we undertake as nothing stays static for very long and constant improvement should always be happening.

That is it. We have come to the end of our introductory journey through the Oracle Warehouse Builder. Hopefully, you have enjoyed it and will take what you've learned and put it to good use in the world of data warehousing with the Oracle Warehouse Builder.

Download at Boykma.Com

Index

Download at Boykma.Com

Download at Boykma.Com

Download at Boykma.Com

Download at Boykma.Com

Download at Boykma.Com

Download at Boykma.Com

Download at Boykma.Com

Download at Boykma.Com

Download at Boykma.Com

Download at Boykma.Com

Thank you for buying

Oracle Warehouse Builder 11g

Getting Started

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

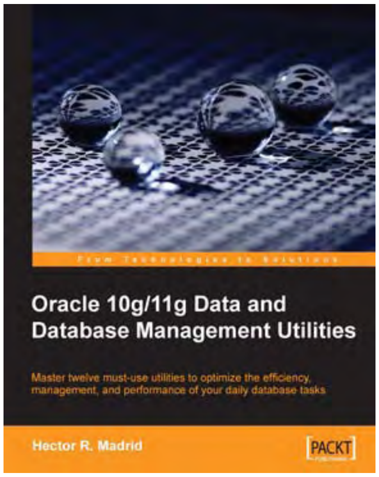
Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

Download at Boykma.Com

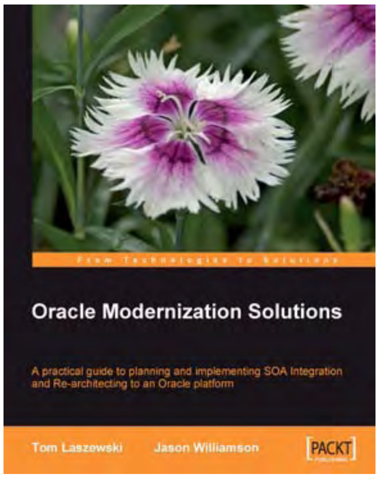


Oracle 10g/11g Data and Database Management Utilities

ISBN: 978-1-847196-28-6 Paperback: 432 pages

Master twelve must-use utilities to optimize the efficiency, management, and performance of your daily database tasks

1. Optimize time-consuming tasks efficiently using the Oracle database utilities
2. Perform data loads on the fly and replace the functionality of the old export and import utilities using Data Pump or SQL*Loader
3. Boost database defenses with Oracle Wallet Manager and Security
4. A handbook with lots of practical content with real-life scenarios



Oracle Modernization Solutions

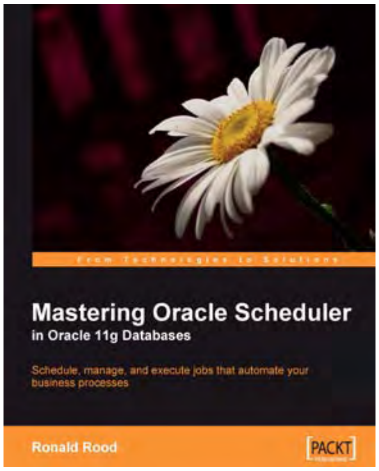
ISBN: 978-1-847194-64-0 Paperback: 432 pages

A practical guide to planning and implementing SOA Integration and Re-architecting to an Oracle platform

1. Complete, practical guide to legacy modernization using SOA Integration and Re-architecture
2. Understand when and why to choose the non-invasive SOA Integration approach to reuse and integrate legacy components quickly and safely
3. Understand when and why to choose Re-architecture to reverse engineer legacy components and preserve business knowledge in a modern open and extensible architecture

Please check **www.PacktPub.com** for information on our titles

Download at Boykma.Com


From Technologies to Solutions
Mastering Oracle Scheduler
in Oracle 11g Databases
Schedule, manage, and execute jobs that automate your business processes
Ronald Rood
PACKT

Mastering Oracle Scheduler in Oracle 11g Databases

ISBN: 978-1-847195-98-2 Paperback: 240 pages

Schedule, manage, and execute jobs that automate your business processes

1. Automate jobs from within the Oracle database with the built-in Scheduler
2. Boost database performance by managing, monitoring, and controlling jobs more effectively
3. Contains easy-to-understand explanations, simple examples, debugging tips, and real-life scenarios

From Technologies to Solutions
Oracle Web Services Manager
Securing Your Web Services
Sitaraman Lakshminarayanan
PACKT

Oracle Web Services Manager

ISBN: 978-1-847193-83-4 Paperback: 236 pages

Securing your Web Services

1. Secure your web services using Oracle WSM
2. Authenticate, Authorize, Encrypt, and Decrypt messages
3. Create Custom Policy to address any new Security implementation
4. Deal with the issue of propagating identities across your web applications and web services
5. Detailed examples for various security use cases with step-by-step configurations

Please check **www.PacktPub.com** for information on our titles

Download at Boykma.Com