Prof:Siddhesh Zele's

DATA WAREHOUSING UNIT 1,2,3,4,5,6

TYBSC(IT) SEM 6

COMPILED BY : SZ,JP,IN

302 PARANJPE UDYOG BHAVAN, NEAR KHANDELWAL SWEETS, NEAR THANE STATION , THANE (WEST)

PHONE NO: 8097071144 / 8097071155 / 8655081002

UNIT 1

1.1 Introduction to Data Warehouse

What Is A Data Warehouse?

A data warehouse is a powerful database model that significantly enhances the user's ability to quickly analyze large, multidimensional data sets. It cleanses and organizes data to allow users to make business decisions based on facts. Hence, the data in the data warehouse must have strong analytical characteristics. Creating data to be analytical requires that it be subject-oriented, integrated, time-referenced, and non-volatile.

Subject-Oriented Data

In a data warehouse environment, information used for analysis is organized around subjects: employees, accounts, sales, products, and so on. This subject specific design helps in reducing the query response
time by searching through very few records to get an answer to the user's question.

Integrated Data

Integrated data refers to de-duplicating information and merging it from many sources into one consistent location. When short listing your top 20 customers, you must know that "HAL" and "Hindustan Aeronautics Limited" are one and the same. Much of the transformation and loading work that goes into the data warehouse is centered on integrating data and standardizing it.

Time-Referenced Data

Time-referenced data essentially refers to its time-valued characteristic. For example, the user may ask "What were the total sales of product 'A' for the past three years on New Year's Day across region 'Y '?"

Time-referenced data when analyzed can also help in spotting the hidden trends between different associative data elements, which may not be obvious to the naked eye. This exploration activity is
termed "data mining".

Non-Volatile Data

The non-volatility of data, characteristic of data warehouse, enables users to dig deep into history and arrive at specific business decisions based on facts.

Why A Data Warehouse?

The Data Access Crisis

If there is a single key to survival in the 1990s and beyond, it is being able to analyze, plan, and react to changing business conditions in a much more rapid fashion. In order to do this, top

managers, analysts, and knowledge workers in our enterprises, need more and better information. Every day, organizations large and small, create billions of bytes of data about all aspects of their business; millions of individual facts about their customers, products, operations and people. But for the most part, this is locked up in a maze of computer systems and is exceedingly difficult to get at. **This phenomenon has been described as "data in jail".**

Data Warehousing

Data warehousing is a field that has grown from the integration of a number of different technologies and experiences over the past two decades. These experiences have allowed the IT industry to identify the
key problems that need to be solved.

Operational vs. Informational Systems

Operational systems, as their name implies, are the systems that help the every day operation of the enterprise. These are the backbone systems of any enterprise, and include order entry, inventory, manufacturing, payroll and accounting. Due to their importance to the organization, operational systems were almost always the first parts of the enterprise to be computerized.

Informational systems deal with analyzing data and making decisions, often major, about how the enterprise will operate now, and in the future. Not only do informational systems have a different focus from operational ones, they often have a different scope. Where operational data needs are normally focused upon a single area, informational data needs often span a number of different areas and need large amounts of related operational data.

Framework Of The Data Warehouse

One of the reasons that data warehousing has taken such a long time to develop is that it is actually a very comprehensive technology. In fact, it can be best represented as an enterprise-wide framework for managing informational data within the organization. In order to understand how all the components involved in a data warehousing strategy are related, it is essential to have a Data Warehouse Architecture.

Data Warehouse Architecture

A Data Warehouse Architecture (DWA) is a way of representing the overall structure of data, communication, processing and presentation that exists for end-user computing within the enterprise. The architecture is made up of a number of interconnected parts:

- Source system
- Source data transport layer
- Data quality control and data profiling layer
- Metadata management layer
- Data integration layer
- Data processing layer
- End user reporting layer

Source System

Operational systems process data to support critical operational needs. In order to do this, operational databases have been historically created to provide an efficient processing structure for a relatively small number of well-defined business transactions.

Clearly, the goal of data warehousing is to free the information locked up in the operational systems and to combine it with information from other, often external, sources of data. Increasingly, large organizations are acquiring additional data from outside databases. This information includes demographic, econometric, competitive and purchasing trends. The so-called information superhighway is providing access to more data resources every day.

Clearly, the goal of data warehousing is to free the information locked up in the operational systems and to combine it with information from other, often external, sources of data. Increasingly, large organizations are acquiring additional data from outside databases. This information includes demographic, econometric, competitive and purchasing trends. The so-called information superhighway is providing access to more data resources every day.

Source Data Transport Layer

The data transport layer of the DWA, largely constitutes data trafficking. It particularly represents the tools and processes involved in transporting data from the source systems to the enterprise warehouse system. Since the data volume is huge, the interfaces with the source system have to be robust and scalable enough to manage secured data transmission.

Data Quality Control and Data Profiling Layer

data quality causes the most concern in any data warehousing solution. Incomplete and inaccurate data will jeopardize the success of The data warehouse.

Data warehouses do not generate their own data; rather they rely on the input data from the various source systems. It is very essential to measure the quality of the source data and take corrective action even before the information is processed and loaded into the target warehouse.

Metadata Management Layer

Metadata is the information about data within the enterprise. Record descriptions in a COBOL program are metadata. So are DIMENSION statements in a FORTRAN program, or SQL Create statements. The information in an ERA diagram is also metadata. In order to have a fully functional warehouse, it is necessary to have a variety of metadata available as also facts about the end-user views of data and information about the operational databases. Ideally, endusers should be able to access data from the warehouse (or from the operational databases) without having to know where it resides or the form in which it is stored.

Data Integration Layer

The data integration layer is involved in scheduling the various tasks that must be accomplished to integrate data acquired from various source systems. A lot of formatting and cleansing activities happen in this layer so that the data is consistent across the enterprise. This layer is heavily driven by off-the-shelf tools and consists of high-level job control for the many processes (procedures) that must occur to keep the data warehouse Up-to-date.

Data Processing Layer

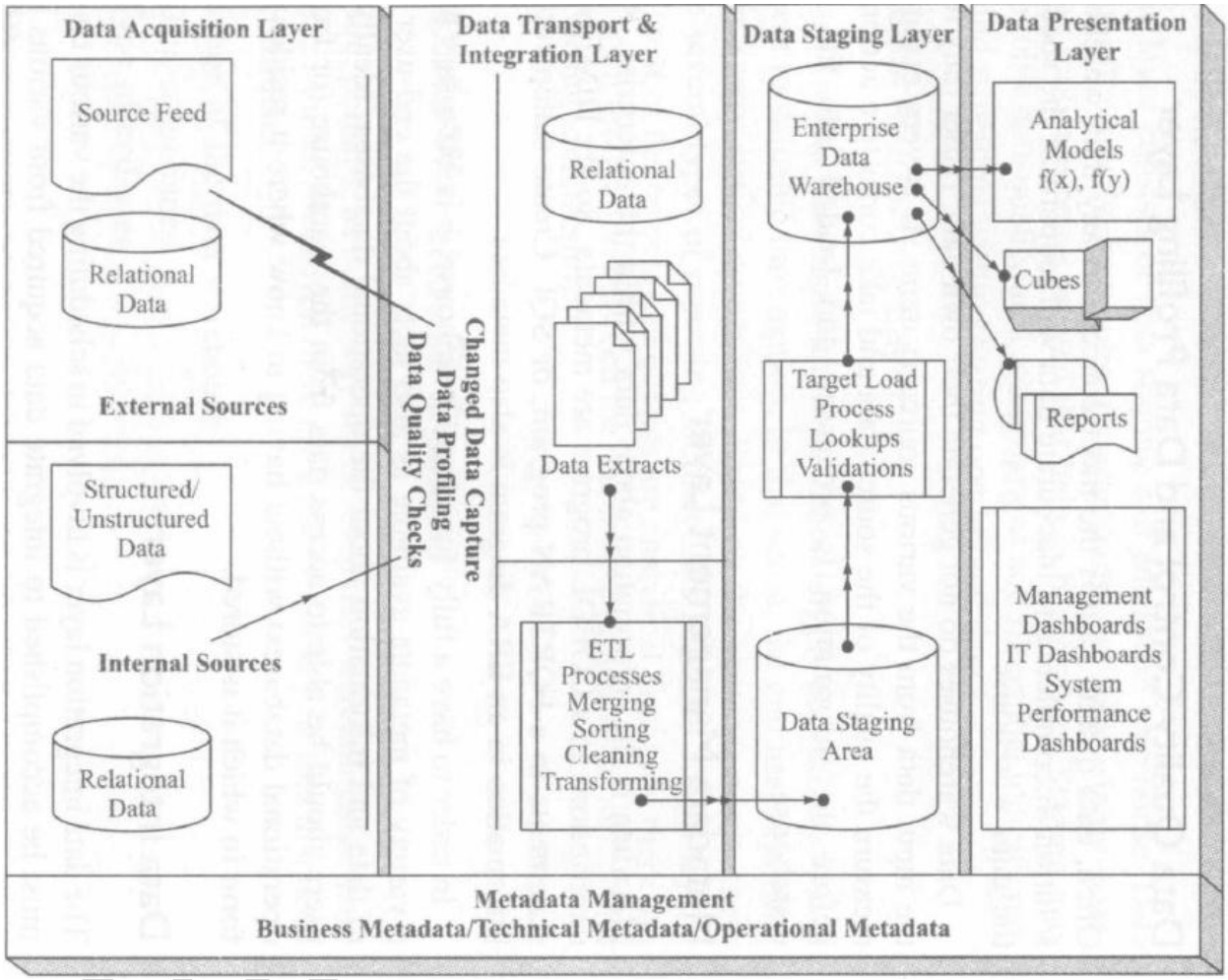
The warehouse (core) is where the dimensionally modeled data resides. In some cases, one can think of the warehouse simply as a transformed view of the operational data; but modeled for analytical purposes.

This layer consists of data staging and enterprise warehouse. Data staging often involves complex programming, but increasingly warehousing tools are being created that help in this process. Staging

may also involve data quality analysis programs and filters that identify patterns and structures within existing operational data.

End User Reporting Layer

Success of a data warehouse implementation largely depends upon ease of access to valuable information. In that sense, the end user reporting layer is a very critical component. Based on the business needs, there can be several types of reporting architectures and data access layers.



Data Warehouse Options

Key factors that need to be considered:

- Scope of the data warehouse
- Data redundancy
- Type of end-user

Scope

The scope of a data warehouse may be as broad as all the informational data for the entire enterprise from the beginning of time, or it may be as narrow as a personal data warehouse for a single

manager for a single year. There is nothing that makes one of these more of a data warehouse than another.

In practice, the broader the scope, the more valuable the warehouse is to the enterprise and the more expensive and time consuming it is to create and maintain.

Data Redundancy

There are essentially three levels of data redundancy that enterprises should think about when considering their data warehouse options:

- "Virtual" or "point-to-point" data warehouses
- Central data warehouses
- Distributed data warehouses

"Virtual" or "point-to-point" Data Warehouses

A virtual or point-to-point data warehousing strategy means that endusers are allowed to get at operational databases directly, using whatever tools are enabled to the "data access network". Virtual data warehouses often provide a starting point for organizations to learn what end-users are really looking for.

Central Data Warehouses

The central data warehouse is a single physical database that contains all data for a specific functional area, department, division, or enterprise.

Such warehouses are often selected where there is a common need for informational data and there are large numbers of end-users already connected to a central computer or network.

A central data warehouse may contain records for any specific period of time and usually, contains information from multiple operational systems.

Distributed Data Warehouses

Distributed data warehouses are those in which certain components are distributed across a number of different physical databases. Increasingly, large organizations are pushing decision-making down to lower levels of the organization and this is in turn, pushing the data needed for decision making down (or out) to the LAN or local computer serving the local decision-maker.

Distributed data warehouses usually involve the most redundant data and, as a consequence, most complex loading and updating processes

Type of End-user

- Executives and managers
- Power users (business and financial analysts, engineers)
- Support users (clerical, administrative)

Developing Data Warehouses

Developing a good data warehouse is no different from any other IT project— it requires careful planning, requirements definition, design, prototyping and implementation. The first and most important element is a planning process that determines what kind of data warehouse strategy the organization is going to start with.

Developing Strategy

Who is the audience? What is the scope? What type of data warehouse should we build?

There are a number of strategies by which organizations can get into data warehousing.

One way is to establish a “virtual data warehouse” environment. This can be done by:

1. Installing a set of data access, data directory and process management facilities
2. Training the end-users
3. Monitoring how the data warehouse facilities are actually used
4. Based on actual usage, creating a physical data warehouse to support the high-frequency requests

A second strategy is to simply build a copy of the operational data from a single operational system and enable the data warehouse from a series of information access tools. This strategy has the advantage of being both simple and fast.

Ultimately, the optimal data warehousing strategy is to select a user population based on value to the enterprise and analyze their issues, questions and data access needs. Based on these needs, prototype data warehouses are built and populated so the end-users can experiment and modify their requirements.

Evolving DWA

The DWA (Data Warehouse Architecture) is simply a framework for understanding data warehousing and how the components of data warehouse fit together. Only the most sophisticated organizations will be able to put together such architecture the first time out. What the DWA provides then, is a kind of roadmap that can be used to design towards. Coupled with an understanding of the options at hand, the DWA provides a useful way of determining if the organization is moving toward a reasonable data warehousing framework.

Designing Data Warehouses

Designing data warehouses is very different from designing traditional operational systems. For one thing, data warehouse users typically don’t know nearly as much about their wants and needs as operational users. Second, designing a data warehouse often involves thinking in terms of much broader, and more difficult to define, business concepts than does designing an operational system. In this respect, data warehousing is quite close to Business Process Reengineering (BPR).

Managing Data Warehouses

- Data warehouses are not magic— they take a great deal of hard work.
- data warehouses require careful management and marketing.
- A data warehouse is a good investment only if end-users can actually get at vital information faster and cheaper than they are using current technology.
- IT management must understand that if they embark on a data warehousing program, they are going to create new demands upon their operational systems— demands for better data, demands for Consistent data and demands for different kinds of data.

End Points

Data warehousing is growing by leaps and bounds and it is becoming increasingly difficult to estimate what new developments are most likely to affect it. Clearly, the development of parallel DB servers with

improved query engines is likely to be one of the most important. Parallel servers will make it possible to access huge data bases in much less time.

Another new technology is data warehouses that allow for the mixing of traditional numbers, text and multimedia. The availability of improved tools for data visualization (business intelligence) will allow users to see things that could never be seen before.

Data warehousing is not a new phenomenon. All large organizations already have data warehouses, but they are just not managing them. Over the next few years, the growth of data warehousing is going to be enormous with new products and technologies coming out frequently. In order to get the most out of this period, it is going to be important that data warehouse planners and developers have a clear idea of what they are looking for and then choose strategies and methods that will provide them with performance today and flexibility for tomorrow.

Goals

In order to provide information with which users can increase profitability, gain competitive advantage or make better business decisions, the data warehouse must meet the following goals:

Provide Easy Access to Corporate Data

- The user access tools must be easy to use, so that accessing warehouse data will be simple and intuitive
- Access should be graphic so that it will be easier for users to understand and spot trends or area of interest
- Access should be manageable by end users of data warehouse, for these are predominantly business analysts, mid-level managers, and CEOs who seldom possess technical acumen to get the data but are highly qualified to analyze the information. They must easily get answers to their questions and ask new questions, all without getting the IT team involved
- The process of getting and analyzing data must be fast. Questions leapfrog, so you must get answers fast. The very nature of data analysis is that not all questions are known beforehand. This involves a lot of unpredictable, ad-hoc inquiry during a typical data analysis session. In an analytic environment, the user is directly interacting with the facts to find patterns, clues or problem areas, rather than looking at a printed report. The answers have to be delivered fast before users lose their train of thought

Provide Clean and Reliable Data for Analysis

- For consistent analysis, the data environment must be stable. Now that the users can create their own reports, they must have consistent data. One department doing an analysis must get the same result as any other.
- Source conflicts must be resolved. The source for warehouse data is the transactional system and it is frequently seen that these systems often have data stored in several different applications in different formats. A custom er ID may be represented as " 100044629" in one system and "44629" in another. A straight load of information for this company into the data warehouse would result in two customers being represented with transactions being listed and no supporting details.
- Historical analysis must be possible, so that data can be analyzed cross a span of time.

1.2 Data Warehouse Design Consideration and Dimensional Modeling

Data Warehouses support business decisions by collecting, consolidating, and organizing data for reporting and analysis with tools such as online analytical processing (OLAP) and data mining models. Although data warehouses are built on relational database technology, the design of a data warehouse data model and subsequent physical implementation differs substantially from the design of an online transaction processing (OLTP) system.

How do these two systems differ and what design considerations should be kept in mind while designing a data warehouse data model?

OLTP Database	***Data Warehouse Database***
Designed for real-time business transactions and processes.	Designed for analysis o f business measures by subject area, category and attributes.
Optimized for a common and known set o f transactions.	Optimized for bulk loads and large complex, unpredictable queries.
Designed for validation o f data during transactions,	Designed to be loaded with consistent, valid data; uses very minimal validation
Supports few concurrent users relative to the OLTP environment.	Supports large user bases often distributed across geographies.
Houses very minimal historical data	Houses a mix o f most current information as well as historical data

An OLTP system requires a normalized structure to minimize redundancy, provide validation of input data, and support a high volume of transactional data processing needs. A transaction usually involves business event(s), such as placing an order, posting an invoice payment, a credit/debit transaction and the like. An OLTP model often looks like a spider web of hundreds or even thousands of related tables.

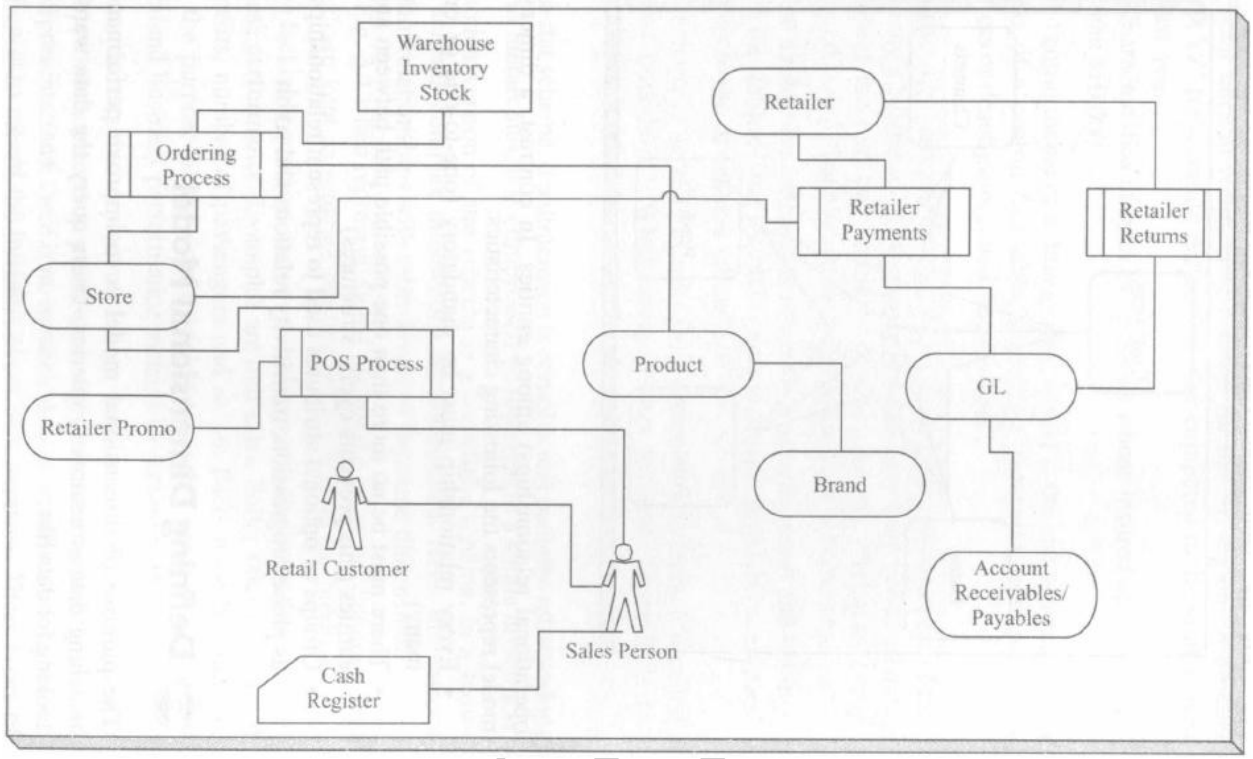
In contrast, a typical dimensional model uses a star or snowflake data model that is easy to understand and relate to business needs, supports simplified business queries, and provides superior data retrieval process by minimizing large table joins.

The principal characteristics of a dimensional model are a set of detailed business facts surrounded by multiple dimensions that describe those facts.

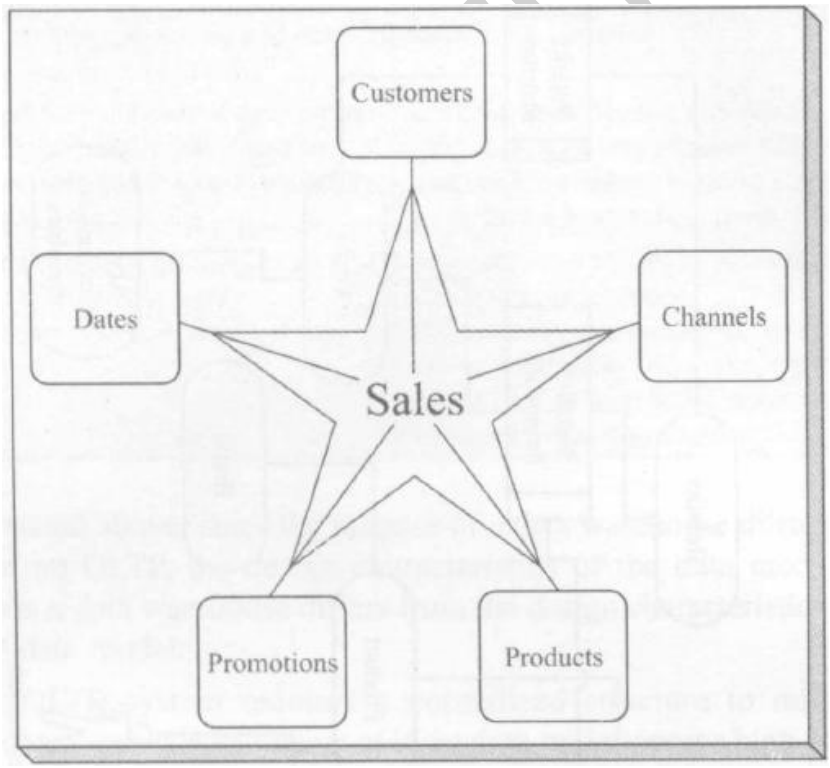
Dimensional models and traditional entity/relationship (E-R) modeling are logical modeling used to capture essential details of the model's subject. E-R diagrams usually represent conceptual models, and are used to capture complex relationships (many-to-many, one-to-many or operational relationships) among entities. In contrast, a dimensional model represents the following characteristics:

- Every relationship must be mandatory (one-to-one or one-to-many)
- There must be no more than one possible path between any two entities (this prevents cyclic structures)
- Groups of optional attributes used to represent relationships such as phase progression, transitivity relations and so on.

OLTP DATA MODEL AND BUSINESS PROCESSES



DIMENSIONAL DATAMODEL AND BUSINESS PROCESSES



Defining Dimensional Model

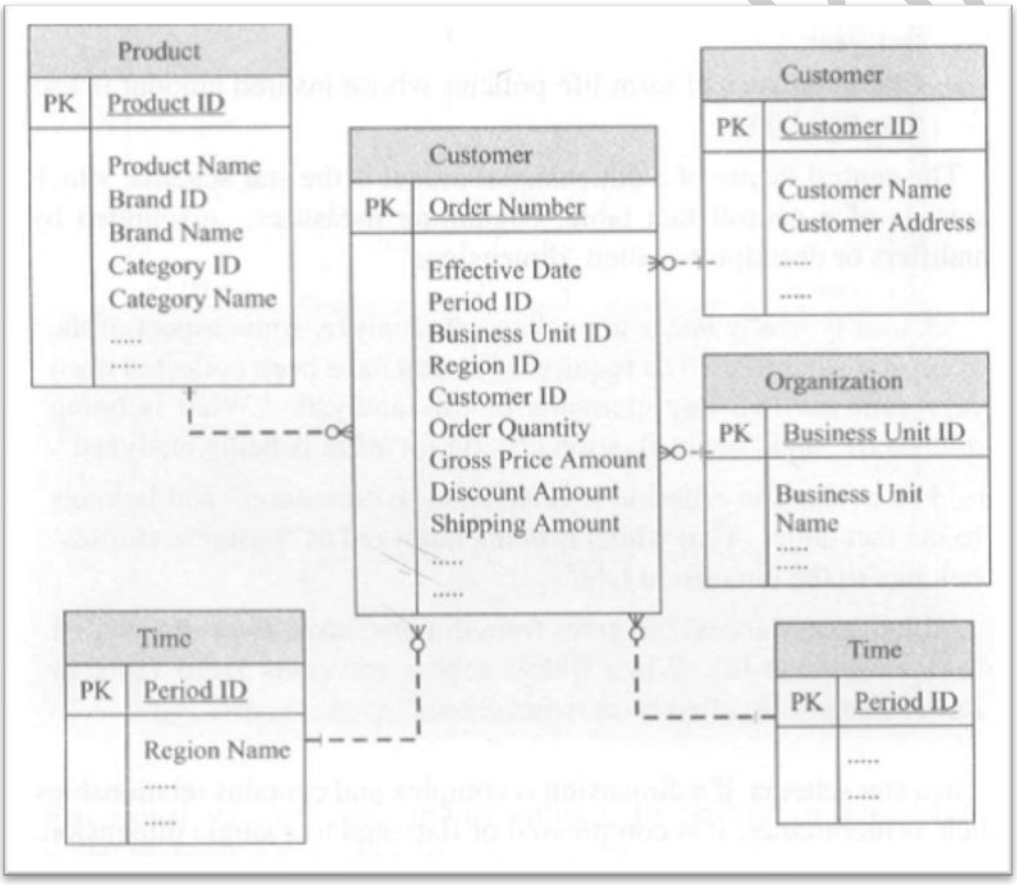
The purpose of dimensional model is to improve performance by matching data structures to queries. Users query the data warehouse looking for data like:

- Total sales in volume and revenue for the NE region for product 'XYZ' for a certain period this year compared to the same period last year
- Characteristics of term life policies whose insured amount is less than $ 10,000

The central theme of a dimensional model is the star schema, which consists of a central fact table, containing measures, surrounded by qualifiers or descriptors called 'dimensions'.

In a star schema, if a dimension is complex and contains relationships such as hierarchies, it is compressed or flattened to a single dimension.

Star Schema Model

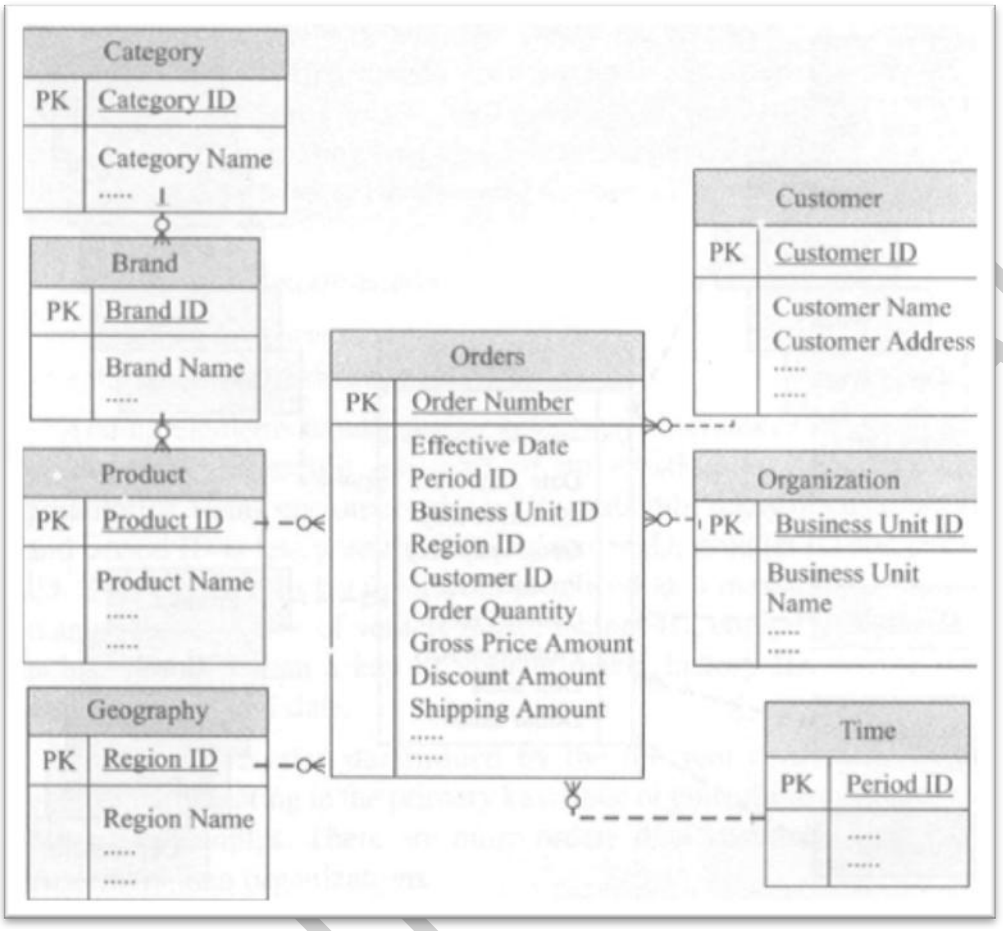


Another version of star schema is a snowflake schema. In a snowflake schema complex dimensions are normalized. Here, dimensions maintain relationships with other levels of the same dimension.

The dimension tables are usually highly non-normalized structures. These tables can be normalized to reduce their physical size and eliminate redundancy which will result in a snowflake schema. Most

dimensions are hierarchic

Snowflake Model



Granularity Of Facts

The granularity of a fact is the level of detail at which it is recorded. If data is to be analyzed effectively, it must be all at the same level of granularity.

The more granular the data, the more you can do with it. Excessive granularity brings needless space consumption and increases complexity. You can always aggregate details.

Granularity is not absolute or universal across industries. This has two implications. First, grains are not predetermined; second, what is granular to one business may be summarized for another.

Granularity is determined by:

- Number of parts to a key
- Granularity of those parts

Adding elements to an existing key always increases the granularity of the data; removing any part of an existing key decreases its granularity.

Granularity is also determined by the inherent cardinality of the entities participating in the primary key.

Data at the order level is more granular than data at the customer level, and data at the customer level is more granular than data at the organization level.

Additivity Of Facts

A fact is additive over a particular dimension if adding it through or over the dimension results in a fact with the same essential meaning as the original, but is now relative to the new granularity.

A fact is said to be fully additive if it is additive over every dimension of its dimensionality; partially additive if additive over at least one but not all of the dimensions; and non-additive if not additive over any dimension.

In general, facts representing individual business transactions are fully additive, although cumulative totals are semi-additive. Non-additive facts are usually the result of ratio or other calculations. Where possible, these should be replaced with the underlying atomic level of data and the derivation formulae should be left to be handled in the queries.

The values that quantify facts are usually numeric, and are often referred to as measures. Measures are typically additive along all dimensions, such as quantity in a sales fact table. A sum of quantity by customer, product, time, or any combination of these dimensions results in a meaningful value.

Some measures are not additive along one or more dimensions, such as quantity-on-hand in an inventory system or price in a sales system. Some measures can be added along dimensions other than the time dimension; such measures are referred to as semi-additive. For example, quantity-on-hand can be added along the inventory dimension to achieve a meaningful total of the quantity of items on hand in all warehouses at a specific point in time. Along the time dimension, however, an aggregate function, such as average, must be applied to provide meaningful information about the quantity of items on hand.

Measures that cannot be added along any dimension are truly non additive. Queries, reports, and applications must evaluate measures properly according to their summarization constraints. Non-additive measures can often be combined with additive measures to create new additive measures. For example, quantity multiplied by price produces extended price or sale amount, an additive value.

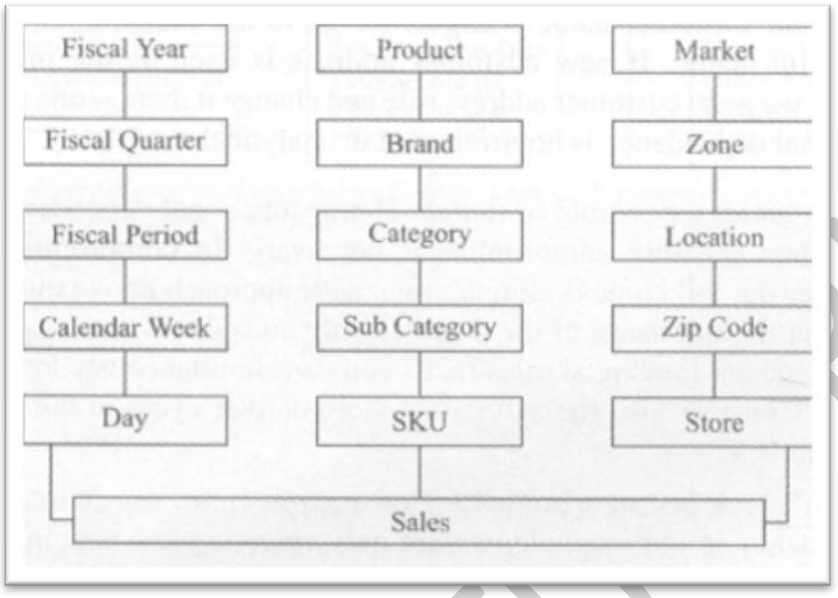
For example, a fact representing cumulative sales units over a day at a store for a product is a fully additive fact. Adding it over any of its dimensions, yields a fact with the same basic meaning except it's for a coarser granularity. However, a fact representing ending inventory on hand for a week at a store for a product is a partially additive fact: Adding it over the dimensions containing the store and product levels doesn't fundamentally change its meaning. But adding over the time dimension provides a result no longer representing ending inventory on hand. The ending inventory on hand for each new sample would have to be the sum of the on-hand values for only those samples corresponding to the latest week in consideration.

Sometimes, careful consideration can provide alternatives to nonadditive facts. For example, a sales fact representing cost is useful. It lets you calculate gross profit. However, per-unit cost is non-additive, which would present some difficulty in summarizing the report data. By storing the cost as an extended cost (per-unit cost times units sold), it will be fully additive. Furthermore, one can use a calculation (sum of extended cost divided by sum of sales units) to obtain average unit cost for any coarser granularity.

Some facts, such as the sales facts in Fig., represent actual events. The actual event can be traced back to at least one business transaction (such as sales, orders, or inventory). The tie to a business transaction may be direct (as with sales and orders, where the fact represents a single aspect of a single

transaction), or indirect (as with the case of the inventory fact, where the fact represents the net effect of some transactions on the state of the business). Hierarchies in Dimensional Model

Hierarchies in Dimensional Model



Other facts represent potential events; those representing the ability of the business to perform a transaction. For example, the set of products a particular store sells during a particular day is typically much smaller than the set of products it carries on that day.

Functional Dependency Of The Data

Functional dependency of data means that the attributes within a given entity are fully dependent on the entire primary key of the entity— no more, no less.

The implication is that each attribute has a one-to-one relationship with its primary key. For each instance of a primary key, there is one set of applicable values; for each attribute, there is only one primary key that properly gives it its identity.

For example, in the customer entity, customer name, date of birth, Social Security Number all belong to the primary key of customer ID. Conversely, if a customer can have different addresses, each with a different role (but only one role), then the customer address depends on, say, customer ID + role. One implication of functional dependency is non-redundancy.

Helper Tables

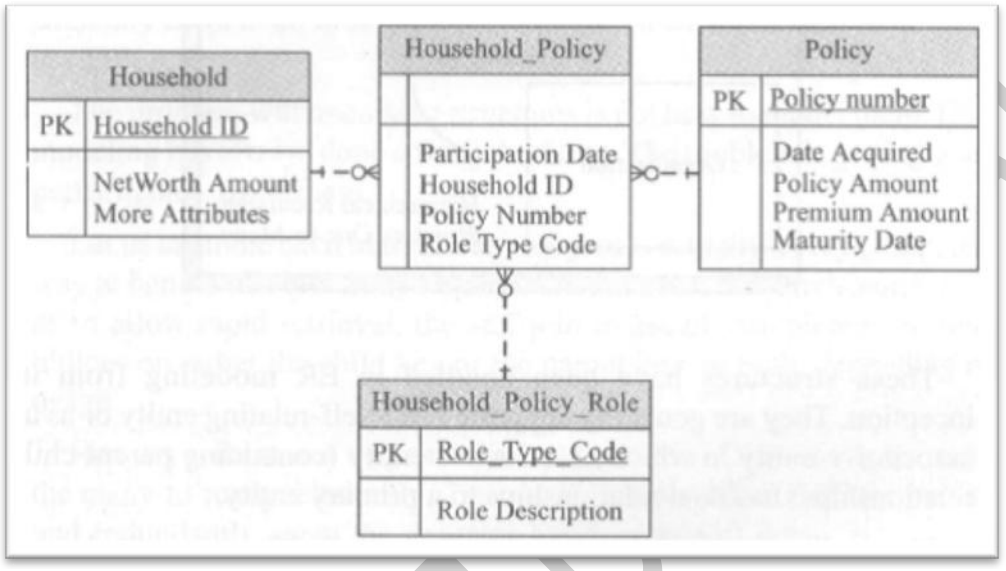
Helper tables usually take one of two forms:

- Help for multi-valued dimensions
- Helper tables for complex hierarchies

Multi-Valued Dimensions

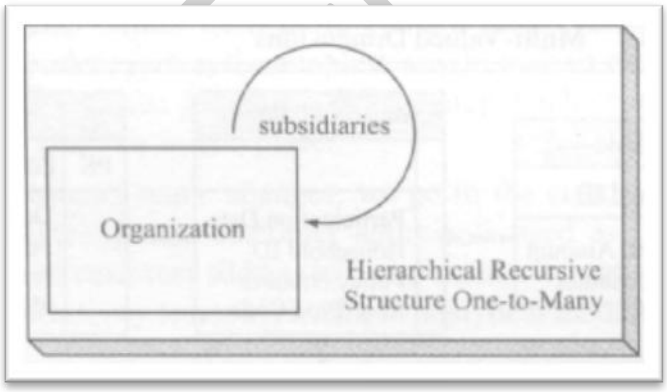
Take a situation where a household can own many insurance policies, yet any policy could be owned by multiple households. The simple approach to this is the traditional resolution of the many-to-many relationship, called an associative entity.

Multi-Valued Dimensions



Complex Hierarchies

A hierarchy is a tree structure, such as an organization chart. Hierarchies can involve some form of recursive relationship. Recursive relationships come in two forms— “self’ relationships (1:M) and “bill of materials” relationships (M: M). A self relationship involves one table whereas a bill of materials involves two.

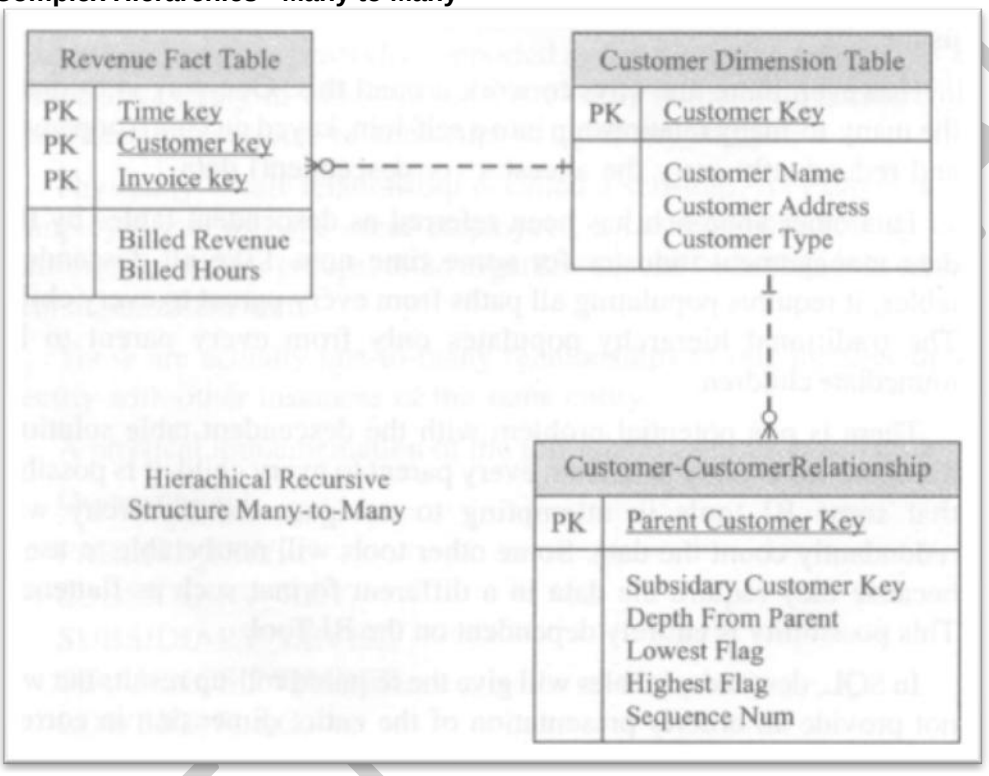


These structures have been handled in ER modeling from its inception. They are generally supported as a self-relating entity or as an associative entity in which an associate entity (containing parent-child relationships) has dual relationships to a primary entity.

Physically, a self-relationship is called a self-join. As examples, an employee can manage other employees, a customer can roll up into different customer groups, or an organization can be broken into different organization units.

These are actually one-to-many relationships of one instance of an entity with other instances of the same entity.

Complex Hierarchies - Many to Many



UNIT 2

2.1 An Introduction to Oracle Warehouse Builder

Introduction to data warehousing

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.

What is a data warehouse?

- It show how companies these days need information to support strategic-level decision making.
- 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.
- 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.
- 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.
- 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.

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 diagram of the database structure has a fact table in the middle surrounded by dimension tables, resulting in something that looks like a star.

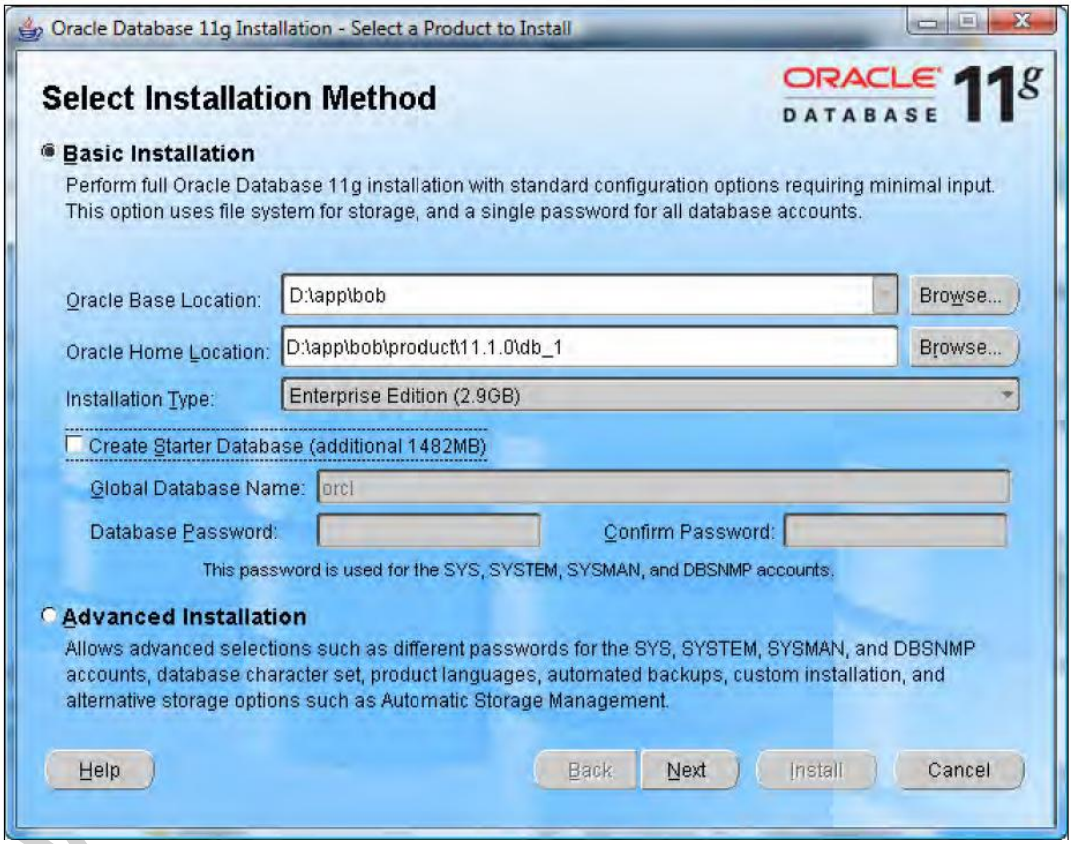
It is also possible that these dimensions may themselves have other tables surrounding them, resulting in something akin to a snowflake.

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.

Installation of the database and OWB

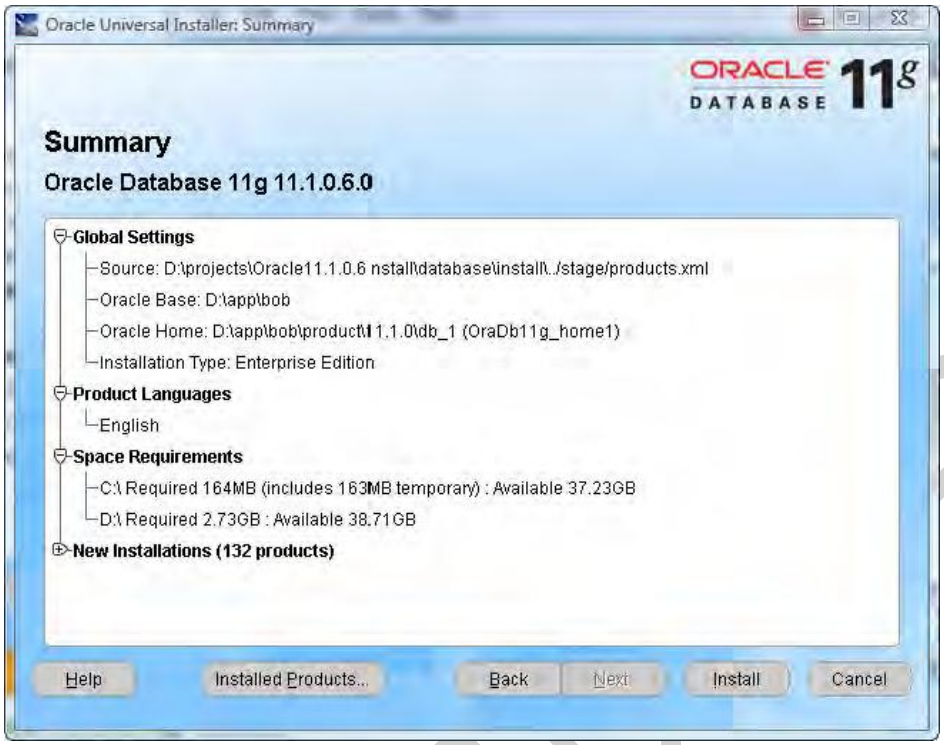
1. run the setup.exe
    - This will launch the Oracle Universal Installer program
    - One of the first questions the installer will ask us is about setting up our ORACLE_HOME—the destination to install the software on the system and the name of the home location.
    - default—OraDb11g_home1



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

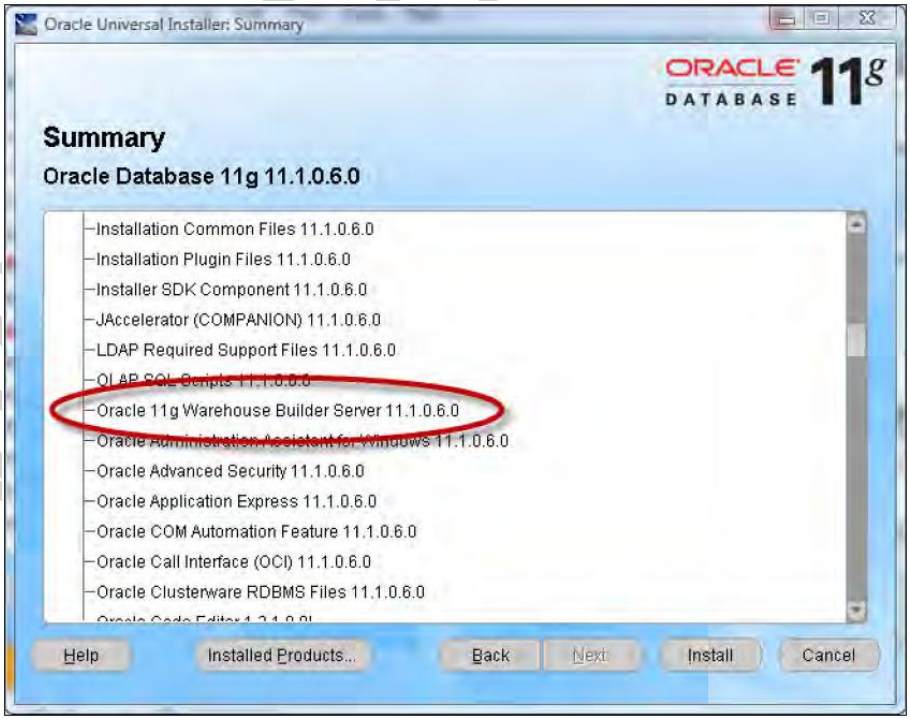
That should show a status of Succeeded for all the checks

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 .



WHAT WE ARE INTRESTED IN

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:

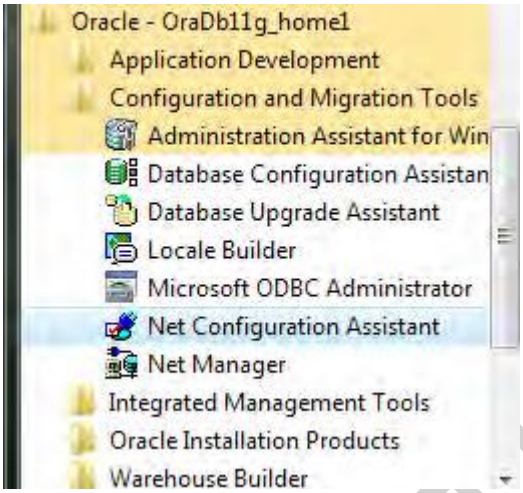


Configuring the listener

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

Run **Net Configuration Assistant** to configure the listener. It is available under the **Oracle** menu on the Windows **Start** menu



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.

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

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

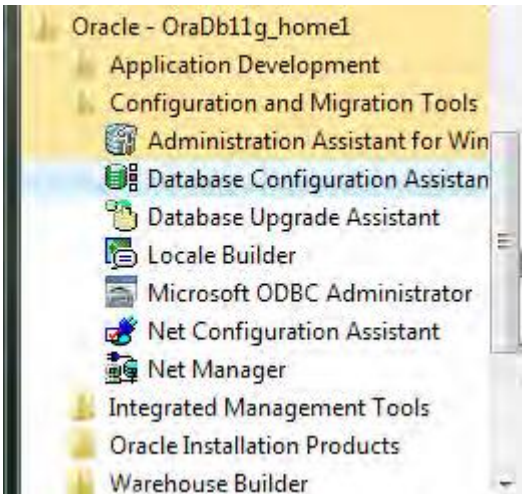
Only the **Add** option will be available since we are installing Oracle for the first time. The remainder of the options will be grayed out and will be unavailable for selection. If they are not, then there is a listener already configured and we can proceed to the next section—*Creating the database*.

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.

Creating the database

We can install a new database using **Database Configuration Assistant**



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.

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

3. This step of the database creation will ask for a database name.

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`.

As the database name is typed in, the **SID** (or Oracle **System Identifier**) is automatically filled in to match it.

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.
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.
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
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.

9. 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.

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.

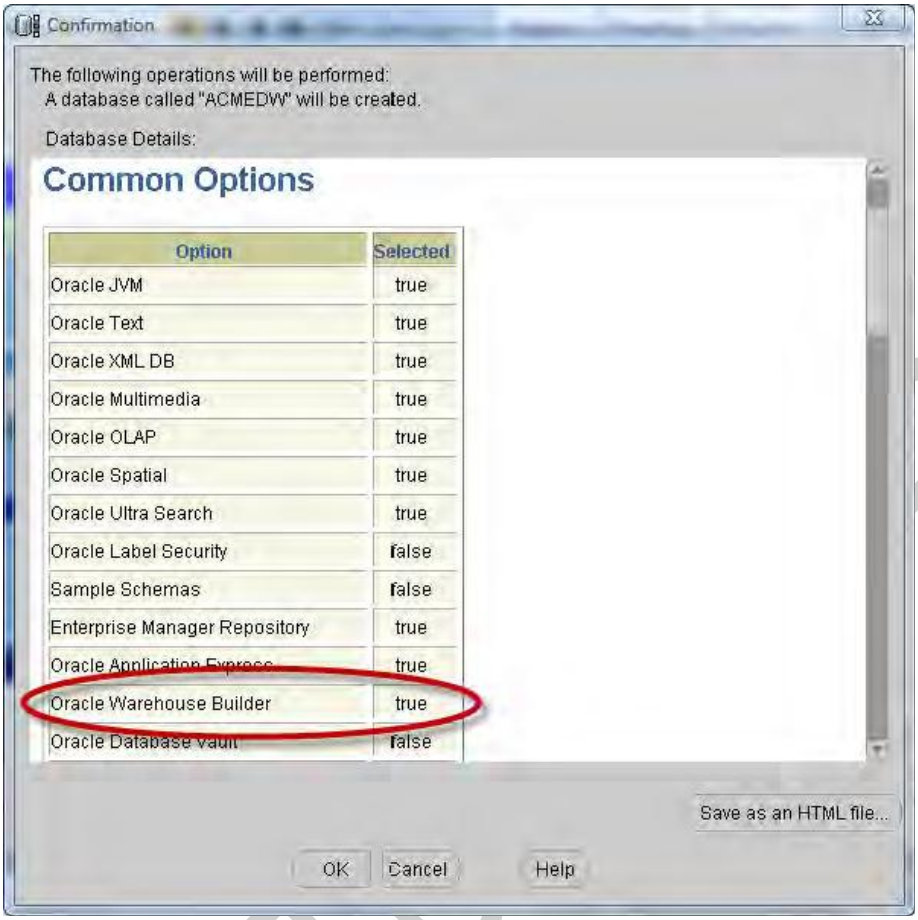
13. The next step (step 13 of 14) is the **Database Storage** screen referred to earlier. Here the locations of the pre-built data and control files can be changed if needed. They should be left set to the default for simplicity since this won't be a production database. For a production environment, 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.

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

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 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:

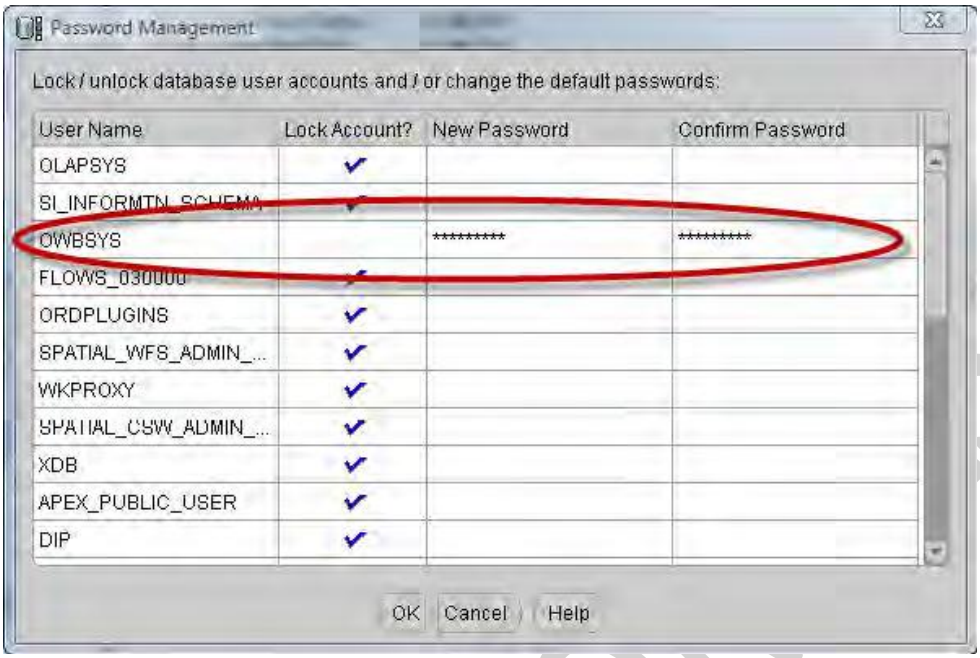


We may see the progress screen at 100% doing nothing with apparently no other display visible. Just look around the desktop underneath other windows for the **Database Configuration Screen**. It's important for the next step

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:

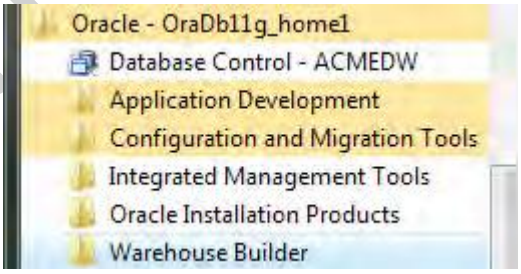


- Click on the **OK** button to apply these changes and close the window. On the Database Configuration Screen, click on the **Exit** button to exit out of the Database Configuration Assistant.

Installing the OWB standalone software:

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:



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.
- So we'll accept it and download the install file to the client computer on which we'll be installing the software.

- 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.

  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.

  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

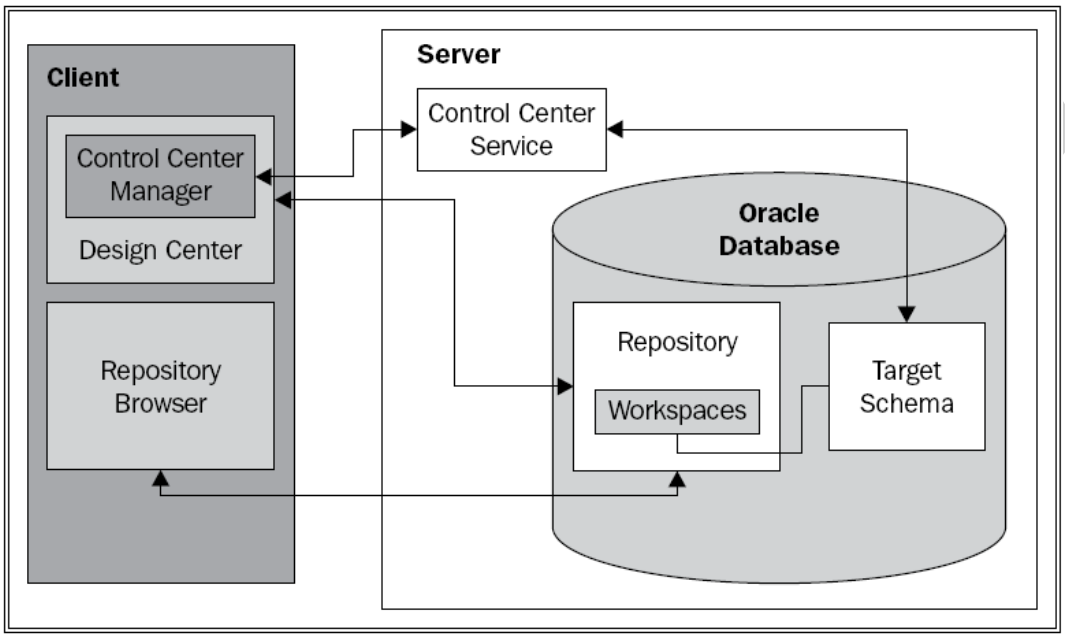
- When installing the standalone OWB software, remember that it cannot be installed into the same ORACLE home as the database.
- It must reside in it own Oracle home folder. So if we have a database that's already installed on the same machine, we'll have to make sure the ORACLE_HOME we specify is different.

1. The next and final step is the summary screen.
2. When the installation is complete, we will be presented with the final success screen and an **Exit** button.

OWB components and architecture:

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



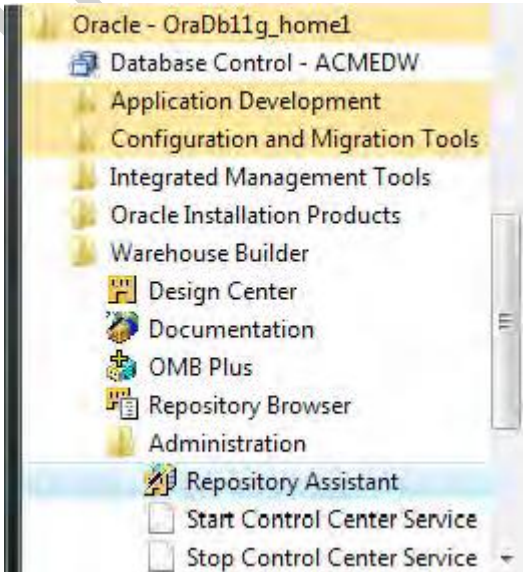
Client and server:

1. These are really just logical notions to indicate the purpose of the individual components and are not necessarily physically separate machines
2. The Design Center is the main client graphical interface for designing our data warehouse.
3. 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.
4. 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
5. Design Center is a logical design only, not a physical implementation.
6. 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.
7. The Control Center Manager for managing the creation of that physical implementation by deploying the designs we've created into the Target Schema..
8. The Control Center Manager to execute the design by running the code associated with the ETL that we've designed.

9. The Control Center Manager interacts behind the scenes with the Control Center Service, which runs on the server as shown in the previous image.
10. The user directly interacts with the Control Center Manager and the Design Center only.
11. 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.
12. 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.
13. 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.
14. The Repository is a Warehouse Builder software component for which a separate schema is created when the database is installed—OWBSYS.
15. 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.
16. For security reasons, we want to restrict user accounts with SYSDBA privilege to a minimum.

Configuring the repository and workspaces:

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:



The most common configuration is to run this application on the same machine where the repository islocated 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.

The steps for configuration are as follows:

1. 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**.

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.

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)))
```

NOTE: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.

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`.

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`
- 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**.

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:

    1. Manage Warehouse Builder workspaces
    2. Manage Warehouse Builder workspace users
    3. Add display languages to repository
    4. Register a Real Application Cluster instance

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. 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.

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.

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.

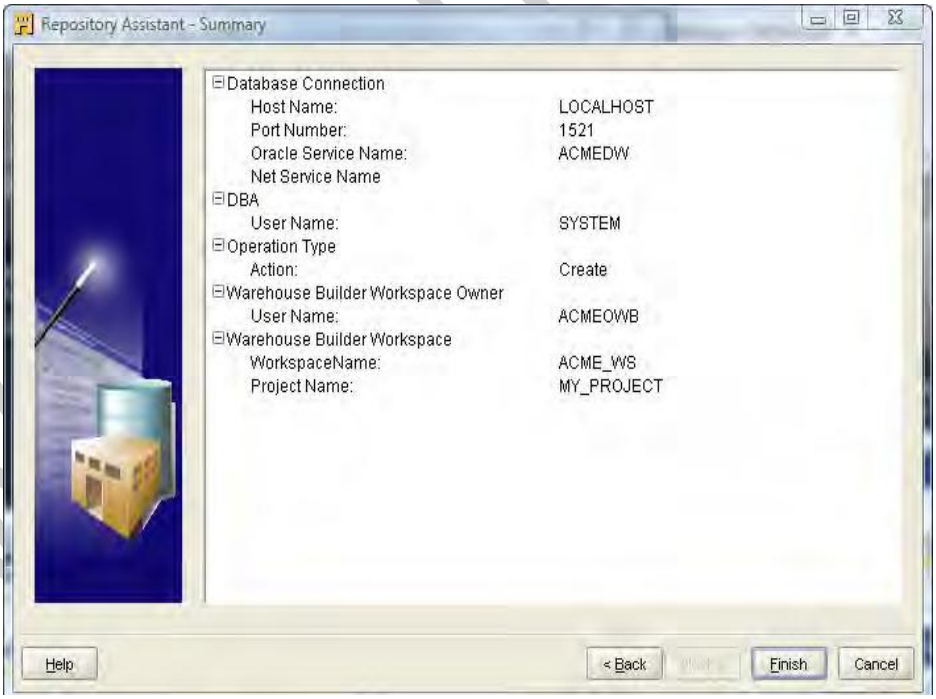
7. This step will ask for the password for the OWBSYS user. This schema was installed for OWB to use for the repository. The password it's looking for is the one we set up back on the final database configuration screen at the end of running **Database Configuration Assistant** to configure the database. 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.

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.

9. This step is to select a base language for the repository, so we'll make the appropriate selection.

10. The final step is the optional step 10 to specify any workspace users from existing database users. The workspace owner is allowed to add and remove database users from the workspace.

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:



11. 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.

2.2 Defining and Importing Source Data Structures

Introduction

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.
- 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.

Preliminary analysis

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.

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 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.

IT department that runs the web site for ACME Toys and Gizmos has its own database , 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

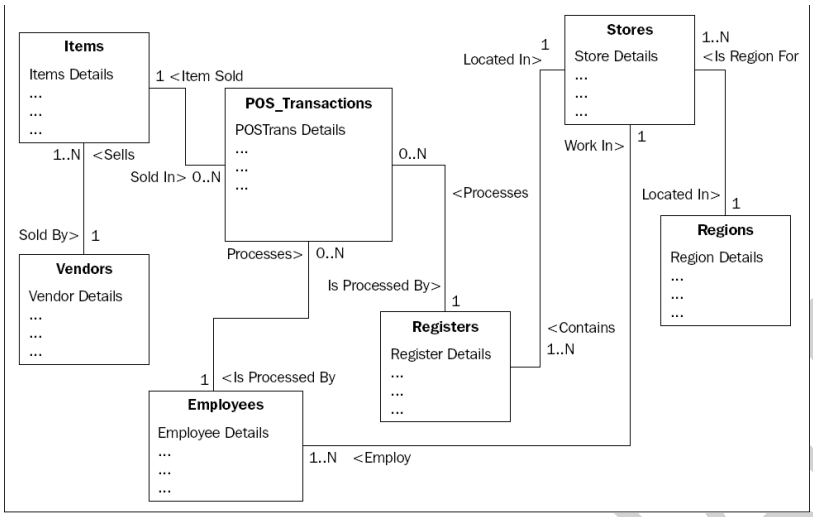
This is an example of the **source data** that might not be needed in the initial stages of a data warehouse project, but that could be requested by users at a later stage.

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 **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:

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



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.

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.

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.

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.

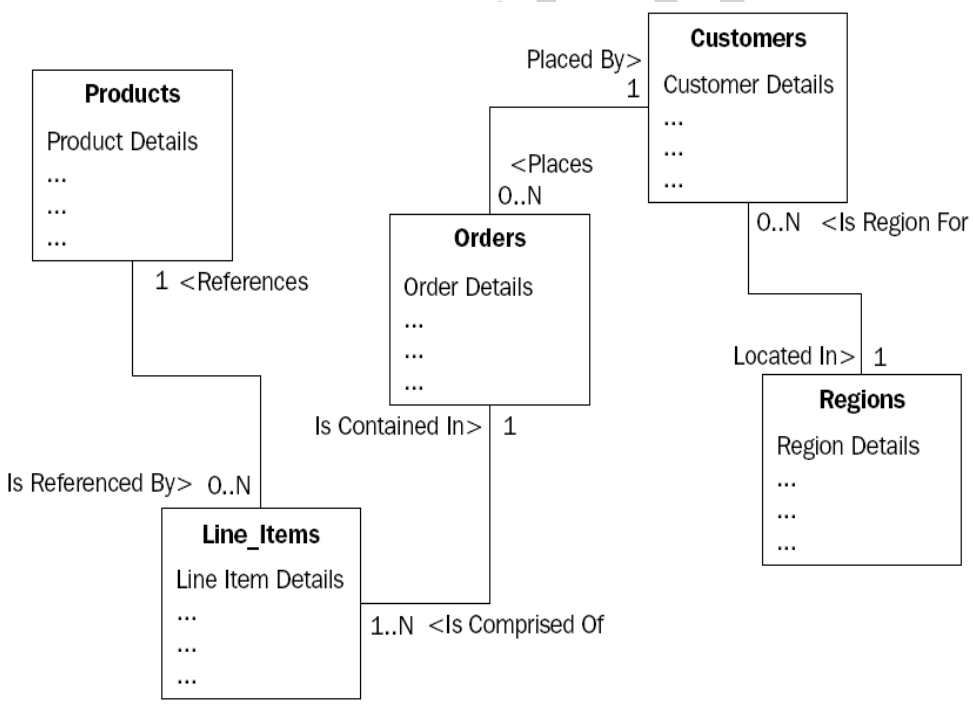
There is no direct connection from the `POS_Transactions` table to the `Regions` table.

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. . 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.

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.

This database holds the sales information for the web site. 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.



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.

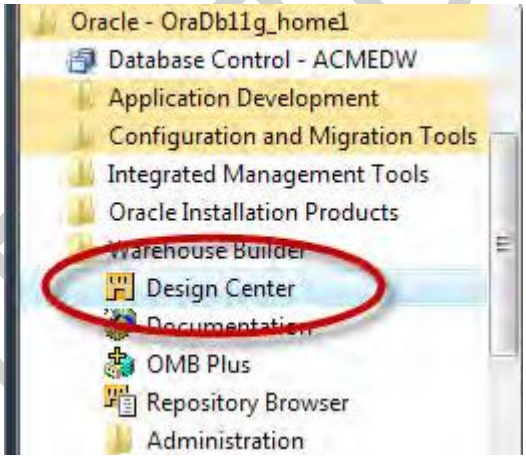
This was a brief overview of the source data structures.

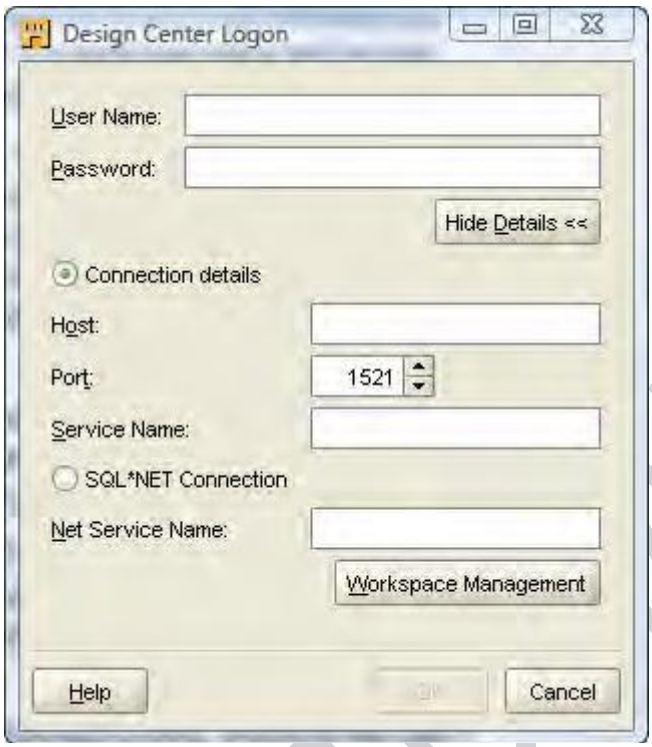
An overview of Warehouse Builder Design Center:

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.

We launch **Design Center** from the **Start** menu under the **Oracle** menu entry, as shown in the following image:

The **Design Center** must connect to a workspace in our repository. 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.



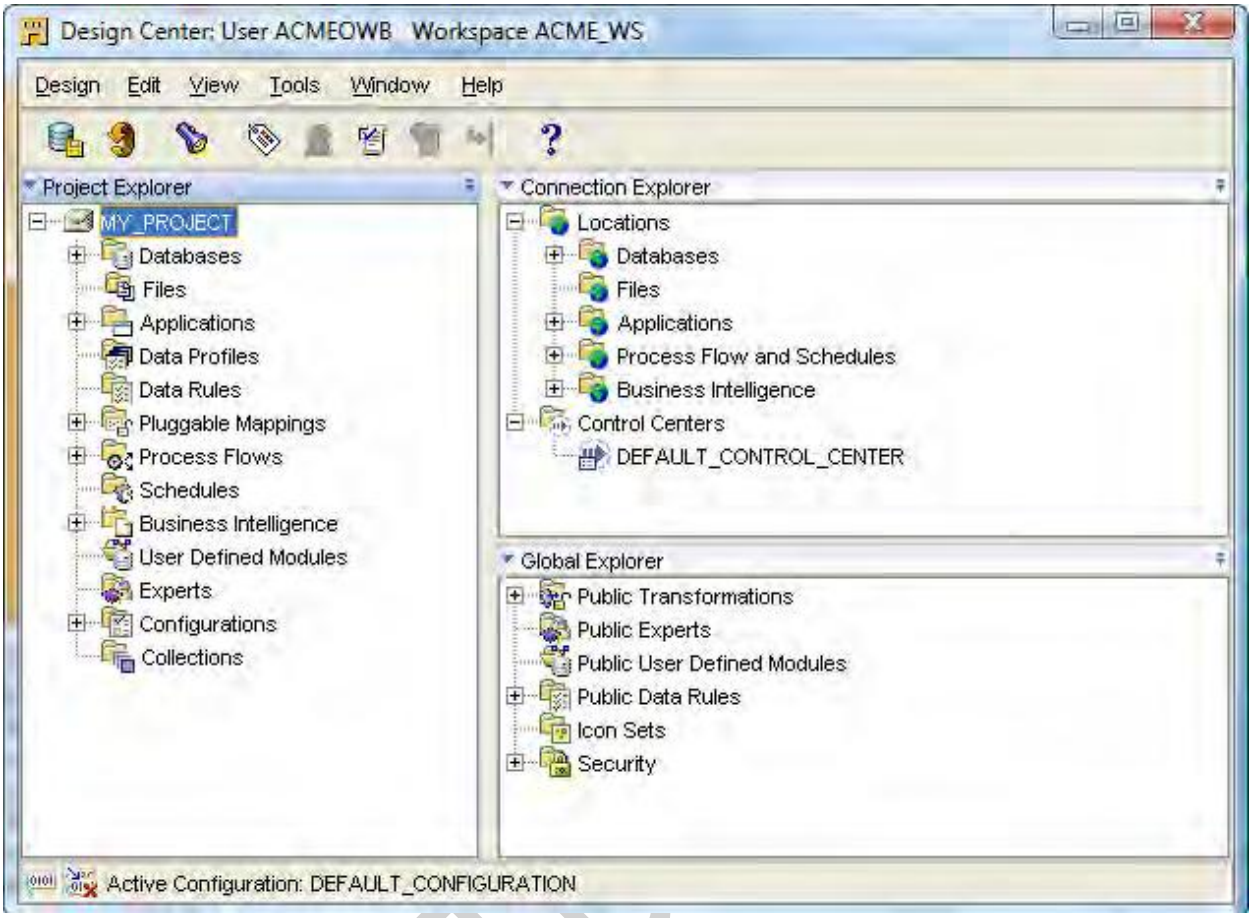


- The first time we use this application, the **Logon** dialog box comes up all blank. But after we fill in our information for the first time, it will remember the **User Name** and **Connection details** on subsequent executions of the Design Center. Also, it will present us with a smaller version 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.

- 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.

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.

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.

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.

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. 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**.

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

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.

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.

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. Launch the **Design Center** now if you haven't already and we'll start working with it.

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.

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.

Creating a module:

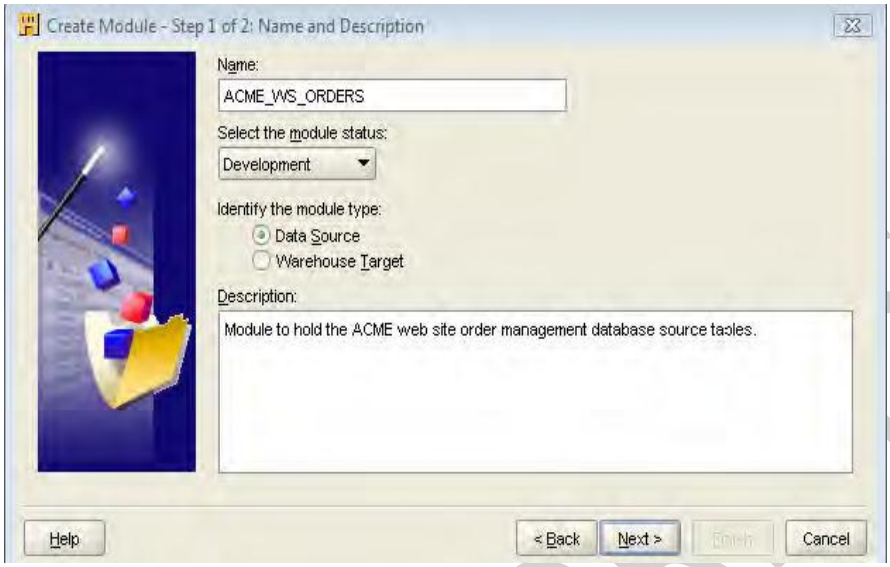
- A **module** is an object in the **Design Center** that acts as a storage location for the various definitions and helps us logically group them
- There are **Files** modules that contain file definitions and **Databases** modules that contain the database definitions. These **Databases** modules are 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.

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.
    - 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.

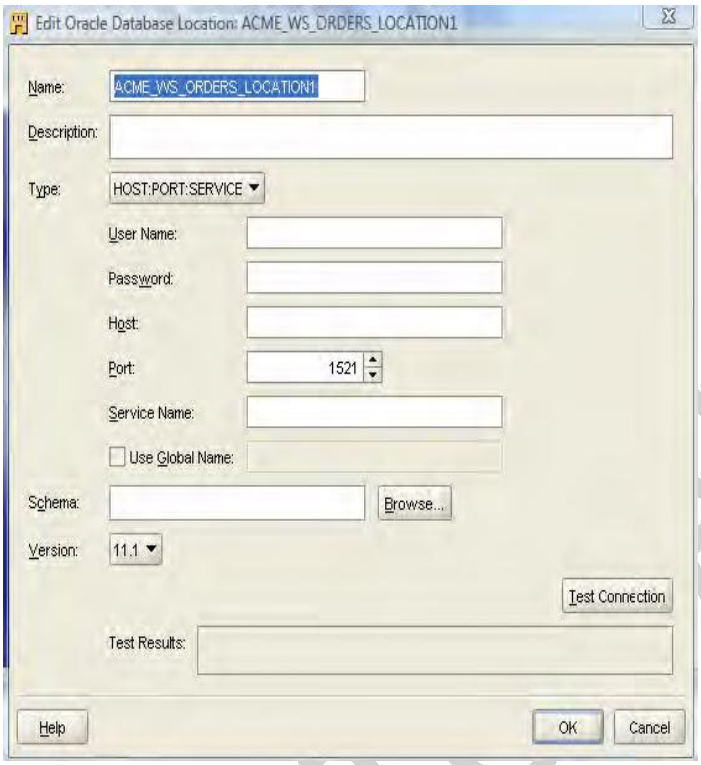
Data Source and our screen should now look similar to the following:



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

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:

1. 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:



2. 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.

3. 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.

4. 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.

5. One final step is to make sure the version of the Oracle database is set correctly.

6. Press the **Test Connection** button and if everything is OK.

7. 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.

8. 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.

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.

Creating a SQL Server database module:

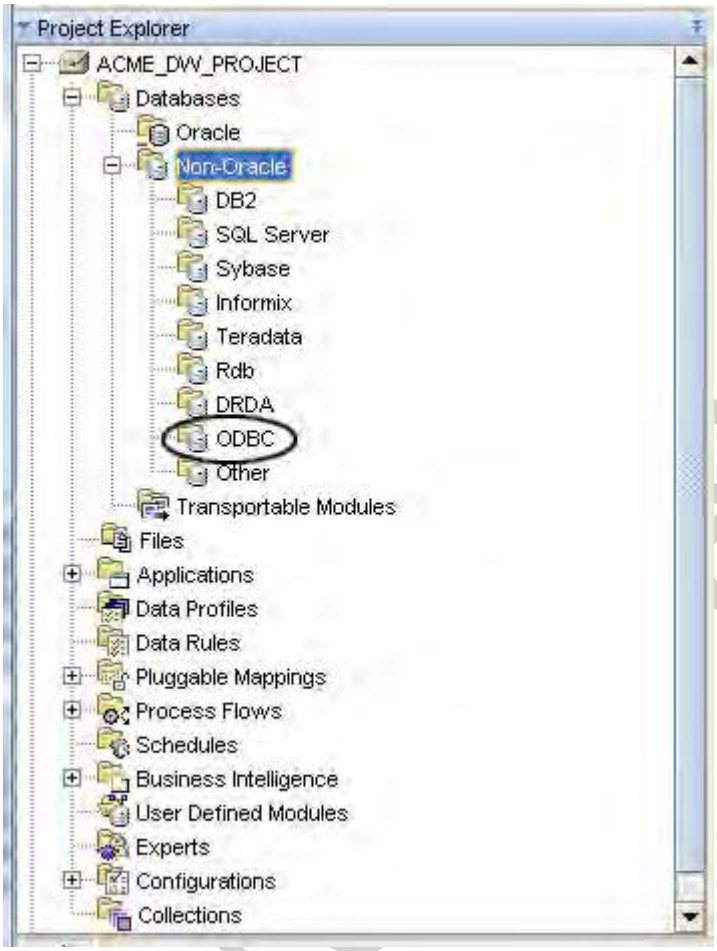
In the following image, a number of databases are listed, including the one that says just **ODBC**:

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.

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.

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.

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.

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.

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 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.

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.

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:

- You can access this application by navigating through the **Start | Control Panel | Administrative Tools** menu. The application is called **Data Sources (ODBC)**.
- In **ODBC Data Source Administrator**, click on the **System DSN** tab, and then click on the **Add** button to add a new system DSN.
- 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.

- For SQL Server, the first screen will ask you for a **Name**, **Description**, and the host on which the SQL Server database is located.

- 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.

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.

- 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.

- 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.

- click on **Next** to continue:

- 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.

- Leave all the other options set as they are and click on the **Next** button to continue.
- click on the **Finish** button to complete the process.
- 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:

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:

- Create a heterogeneous service configuration file.
- 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:

- 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.

- 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>

The lines that begin with # are comment lines and will be ignored.

- 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.

- 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).

- 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.

Editing the listener.ora file:

The steps for this are:

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)
) )
```

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.

Creating the Warehouse Builder ODBC module for SQL Server:

The steps to create an ODBC module and location in Warehouse Builder are:

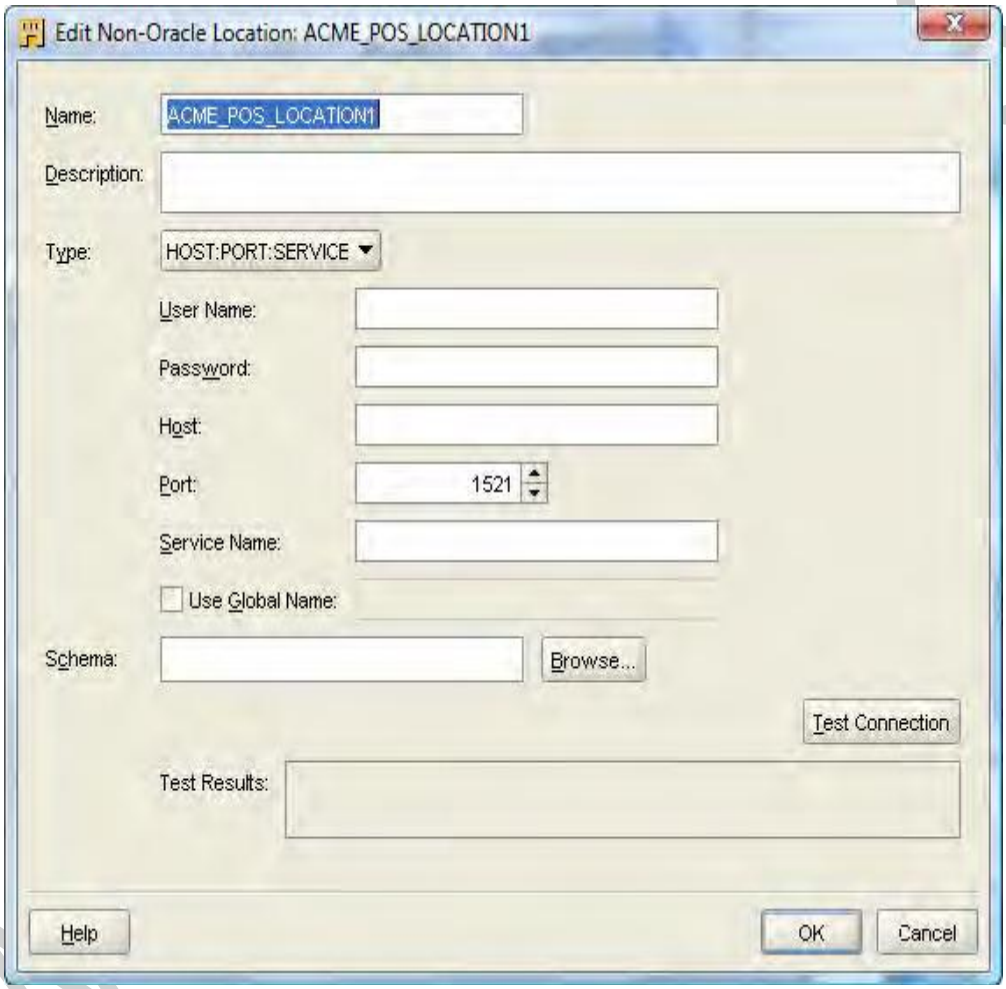
- Right-click on the **ODBC** node in the **Project Explorer** of **Design Center**, and select **New...** from the pop-up menu. The first screen that will appear is the **Welcome** screen, so just click on the **Next** button to continue.

- The screen with the label **Step 1** is where we provide a name as we did for the Oracle module.

- We'll leave **module status** set to **Development**.

- 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.

- 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:



- 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.
- 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.
- 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**.

- 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 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.
- Click on the **OK** button to proceed even if there was an error reported when we clicked on the **Test Connection** button.
- The **Import after finish** checkbox will be checked by default.
- click on the **Finish** button.

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.

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.**

Importing source metadata from a database:

- 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**.
- 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.
- Click on **Next** to move on to **Step 2**:
- Click on the plus sign beside the **Tables** entry to see the complete list of tables to choose from.
- 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...**
- click on the **Finish** button, which will begin the import process.
- During the import, a status dialog box will be displayed showing the progress of the import as each object is imported.

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.
- 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.
- 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

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
- The screen labeled **Step 1** that is used for choosing objects to import will be restricted to only tables and views
- select **Design | Save All** from the toolbar menu of the **Design Center** application.

Defining source metadata manually with the Data Object Editor:

The steps to manually define the source metadata using Data Object Editor are:

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.

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.

- Upon selecting **New**, we are presented with the **Data Object Editor** screen.

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.

- Let's click on the **Columns** tab next and enter the information that describes the columns of the `Items` table.

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)

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 1.0×10^{-130} and 1.0×10^{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

- 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**.

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)

UNIT 3

3.1 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.

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.

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.

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.

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. 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 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.

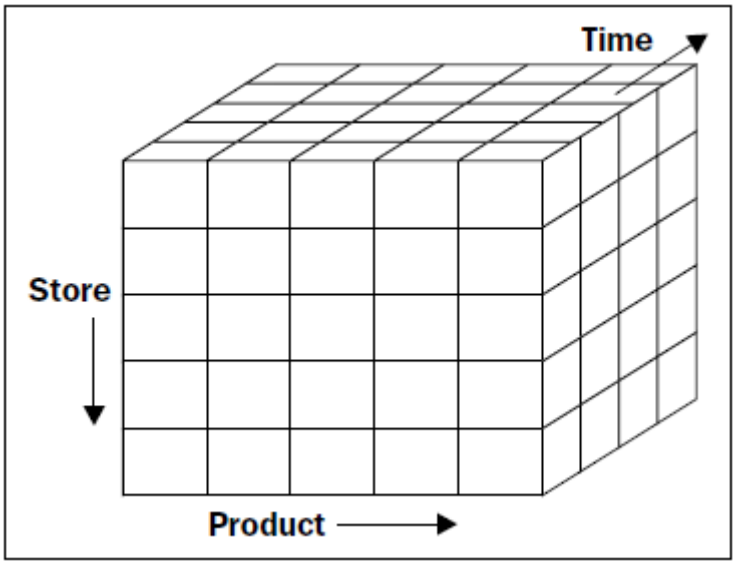
Users can intuitively think of the data for the above question as a cube, and the edges (or dimensions) of the cube labeled as stores, products, and time frame.

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.



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.

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

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.

Relational implementation (star schema)

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. For a relational data warehouse design, the relational characteristics are retained between tables.

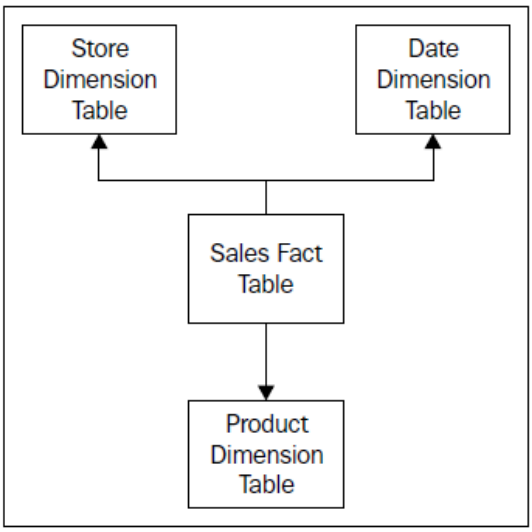
It's much faster and easier to understand if we don't have to include multiple levels of referenced tables. For this reason, a data warehouse dimensional design that is represented relationally in the database will have one main table to hold the primary facts, or **measures** we want to store, such as *count of items sold* or *dollar amount of sales*.

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.

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.

It is the job of data warehouse design to determine what pieces of information need to be included.



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.

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.

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. 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.

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.

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.

Designing the ACME data warehouse

Identifying the dimensions

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.

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.

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. 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.

Designing the cube

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.

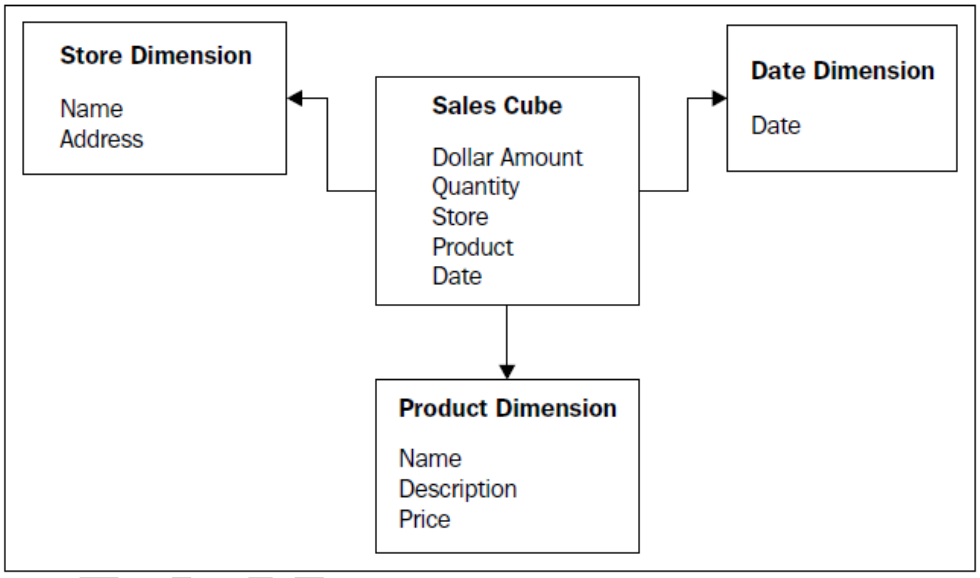
Since we're using sales as the measure, we'll store a sales number; and from our dimensions, we can see that it will be for a given date in a given store and for a given product. Will that number be the total of all the sales for that product for that day? Yes, so it satisfies our design criteria of 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.

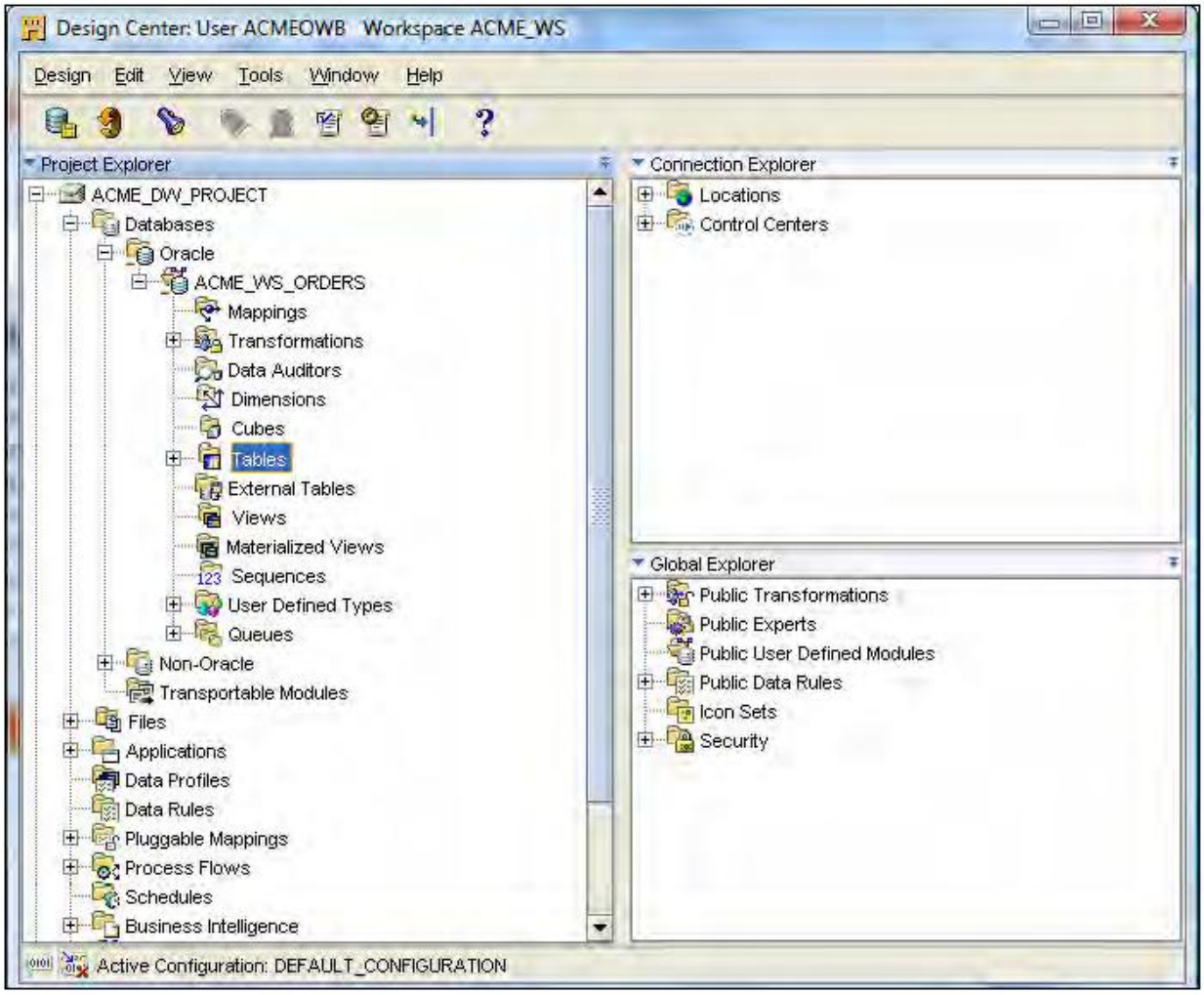
We have found out that the POS Transactional system also maintains the count of the number of a particular item sold in the transaction. This is an additional measure we will consider storing in our cube also, since we can see that it is at the same grain as the total sales. The count of items would 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.



Data warehouse design in OWB

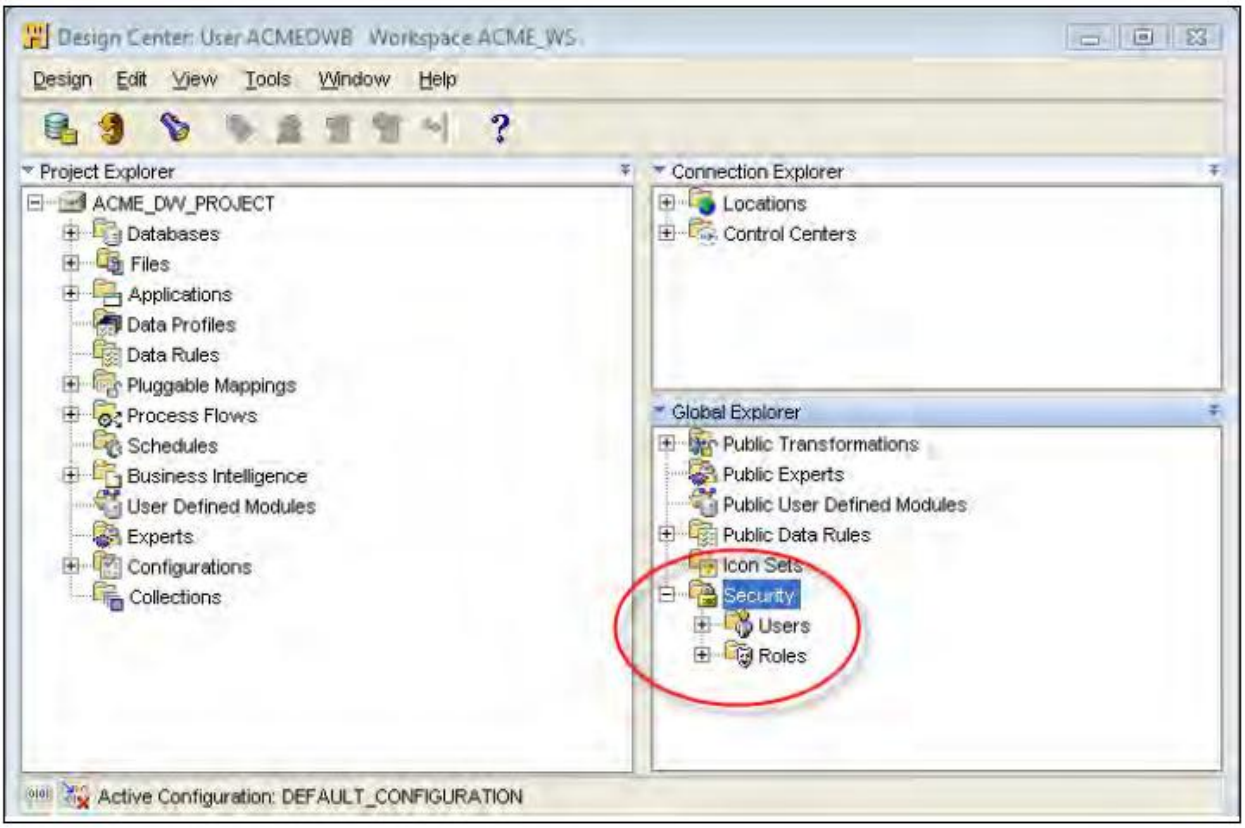
The Warehouse Builder contains a number of objects, which we can use in designing our data warehouse, that are either relational or dimensional. OWB currently supports designing a target schema only in an Oracle database, and so we will find the objects all under the **Oracle** node in the **Project 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. ACME web site orders database—`ACME_WS_ORDERS`.



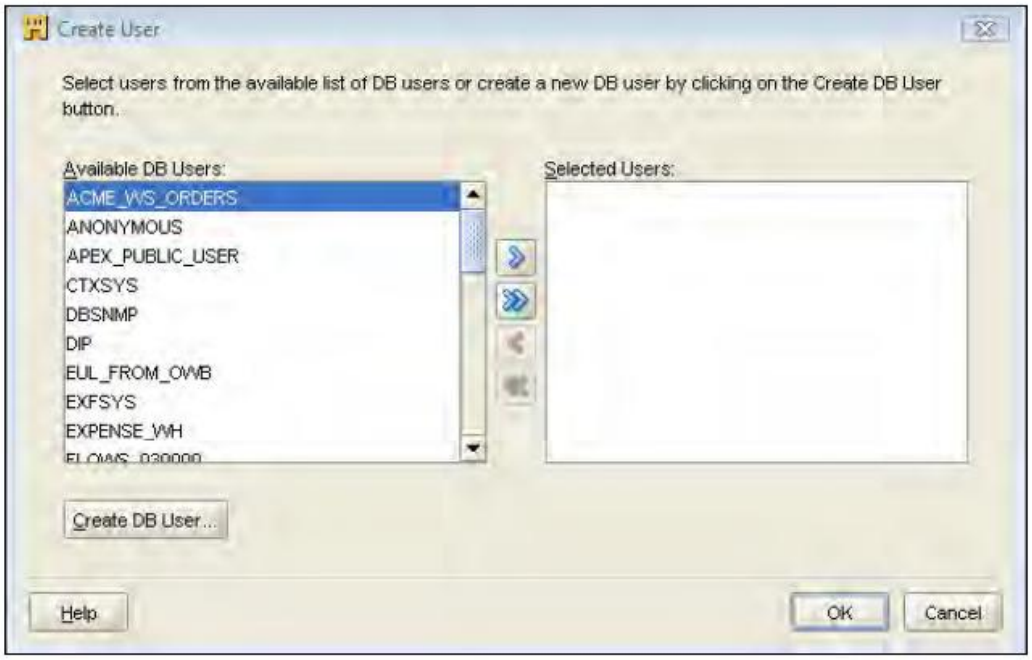
we created the **acmeowb** user as the repository owner and mentioned that this user can be a deployment target for our data warehouse. However, it does not have to be the target user. It's a good idea to create a separate user schema to become the target so that user roles in our database can be kept 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.

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

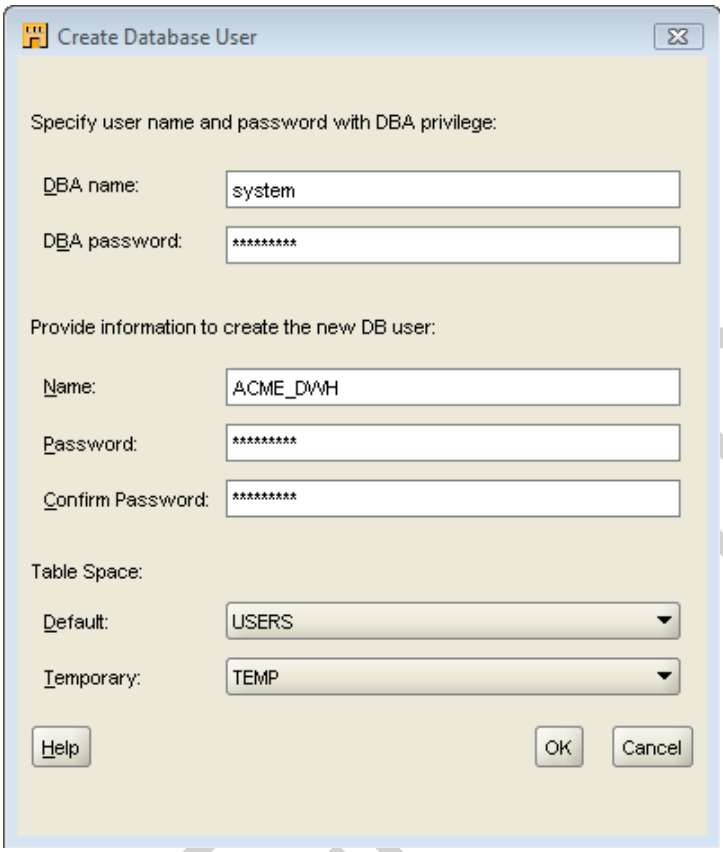


Right-click on the **Users** node and select **New...** to launch the **Create User** dialog box as shown here:



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.

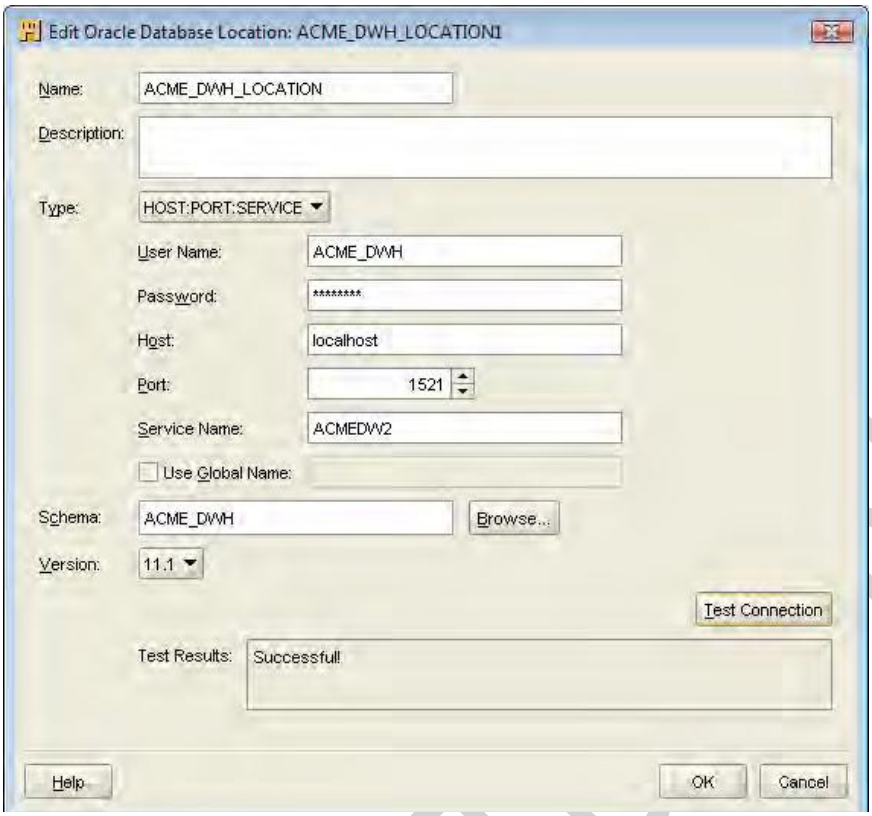


The new user will be created when you click on the **OK** button, and will appear in the righthand window of the **Create User** dialog already selected for us. Click on the **OK** button and the user will be registered with the workspace, and we'll see the new username if we expand the **Users** node 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.

Create a target module

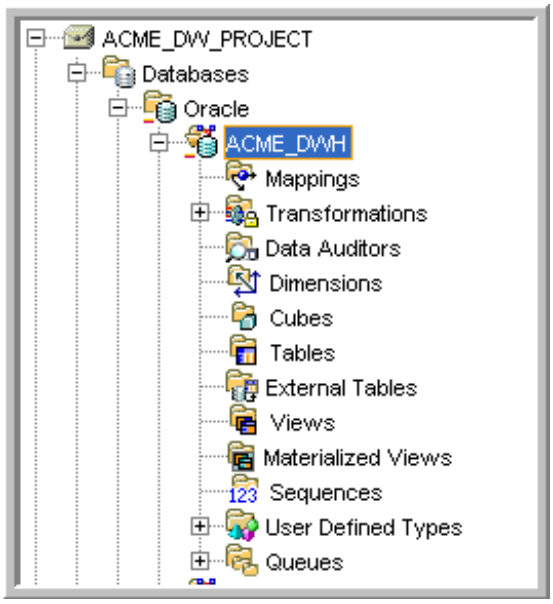
We'll follow the same steps as we did in the last chapter where we created the `ACME_WS_ORDERS` module. Right-click on the **Oracle** object under **Databases** and select **New...** from the pop-up menu to launch the **Create Module Wizard** and step through the process. We'll name this module `ACME_DWH`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



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. 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.

OWB design objects



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. 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.

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. 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**

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. 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 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 provides 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.

3.2 Creating the Target Structure in OWB

Now it's time to actually start creating objects in the Warehouse Builder for our target structure. We'll create the objects using the wizards that the Warehouse Builder provides for us to simplify the task of building cubes and 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.

The Time dimension

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. 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:

- Λεωελσ
- Διμενσιον Αττριβυτεσ
- Λεωελ Αττριβυτεσ
- **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
- Fiscal quarter
- Calendar quarter
- Fiscal year
- Calendar year

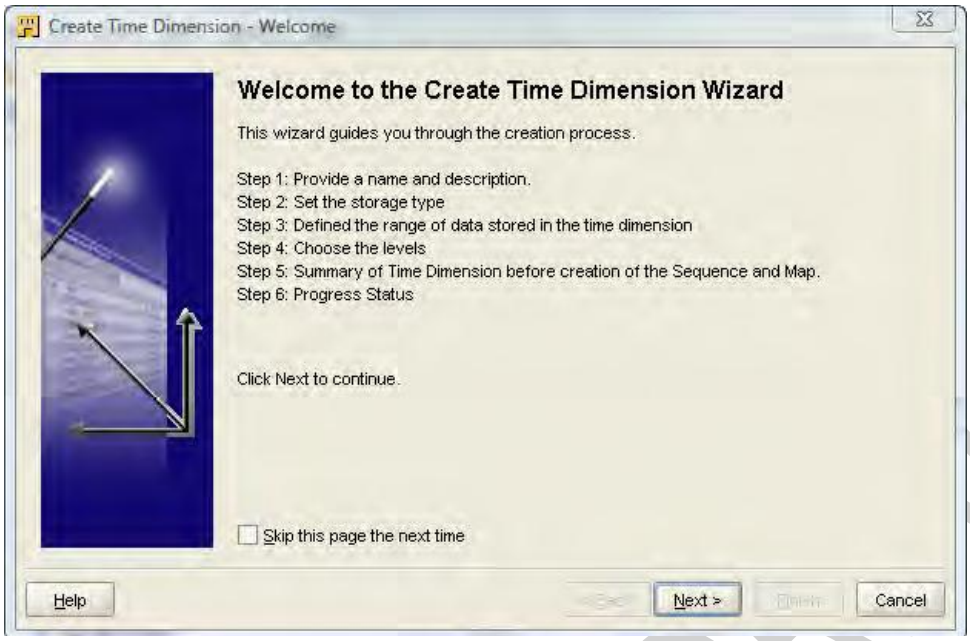
The Dimension Attributes are individual pieces of information we're going to store in the dimension that can be found at more than one level. Each level will have an **ID** that identifies that level, a **start and an end date** for the time period represented at that level, a **time span** that indicates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

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.

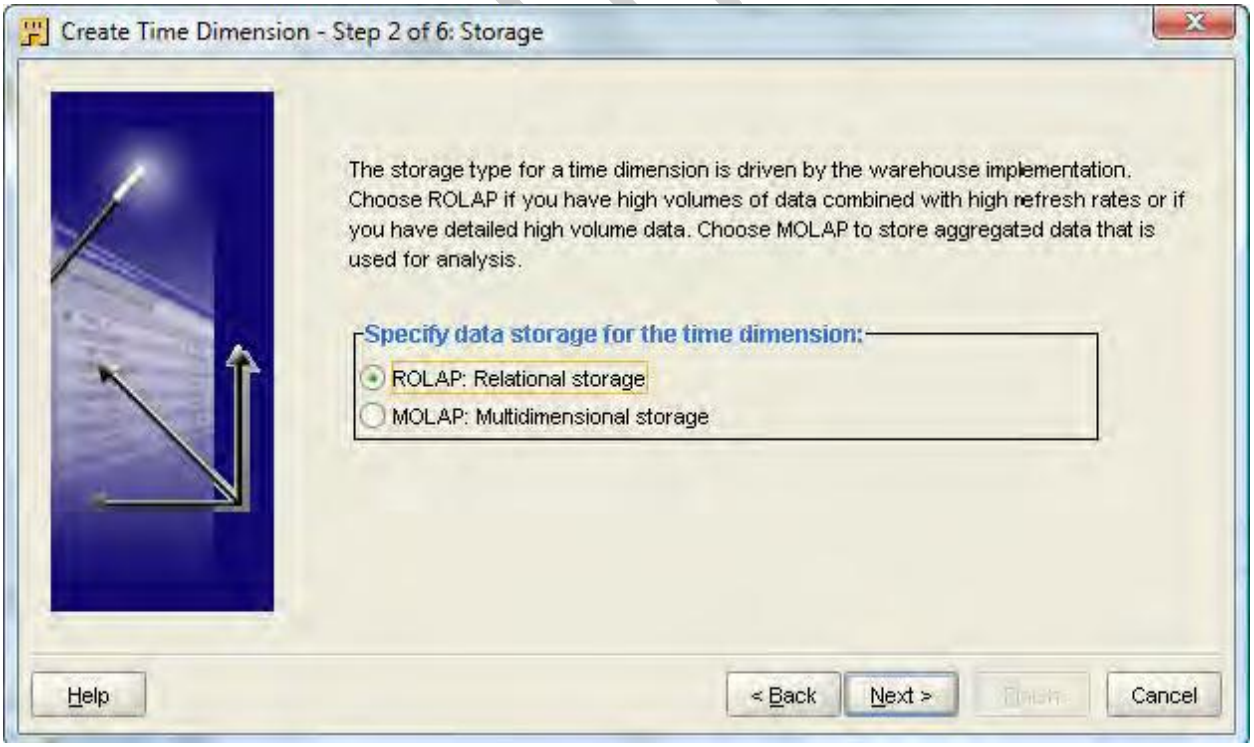
Creating a Time dimension with the Time Dimension Wizard

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.



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.

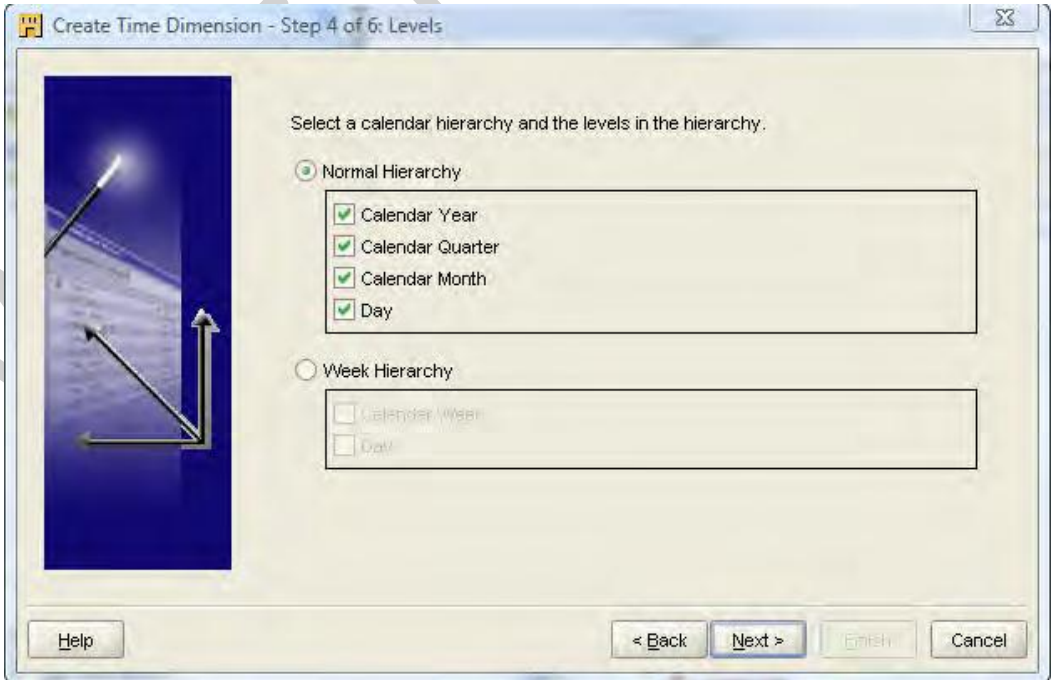
2. The next step will ask us what type of storage to use for our new dimension, as shown here:



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.

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. 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.

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:

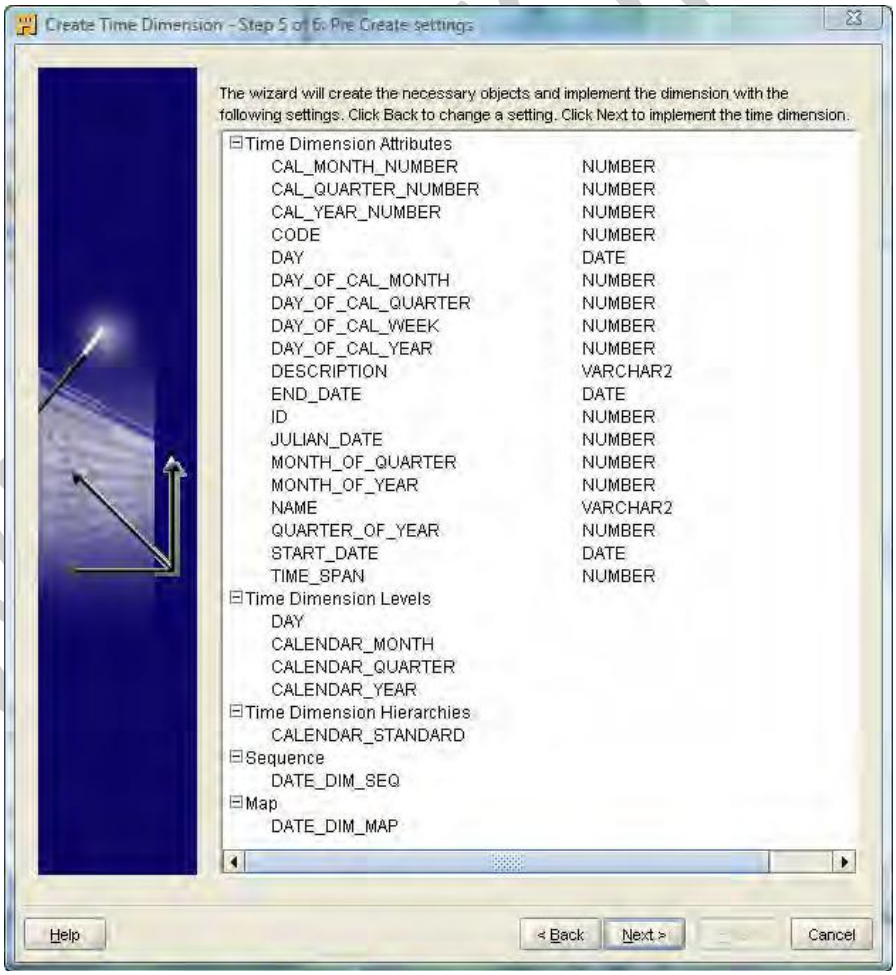


The levels are not available to us because a week does not roll up or aggregate to a month. Some months have four weeks while some have five, and that's not even exact weeks. The only month that has a month evenly divided by weeks is February, and that's only during non-leap years.

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.

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.



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.

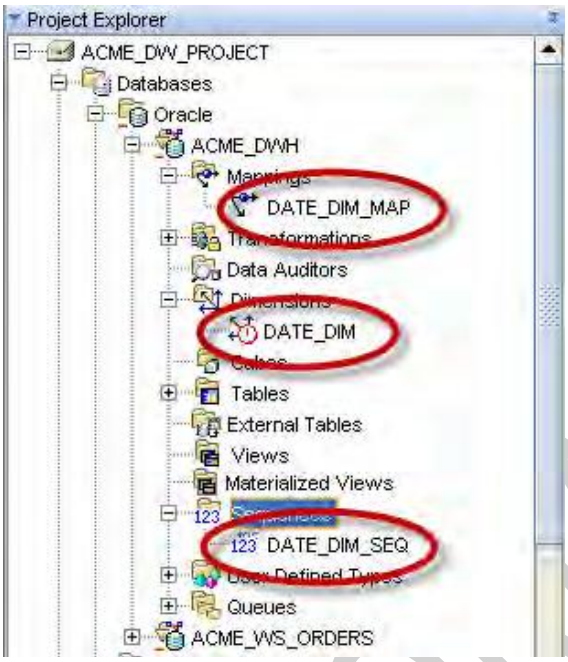
When we link a dimension to a cube, it will use that surrogate identifier as the key to link to, as this is easier for the database to use than a potentially multi-attribute business identifier. However, we think in terms of the business identifier. The Time dimension created by the wizard creates an 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 **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.

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.



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.

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.

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.

Product Attributes (attribute type)

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

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.

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:

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.

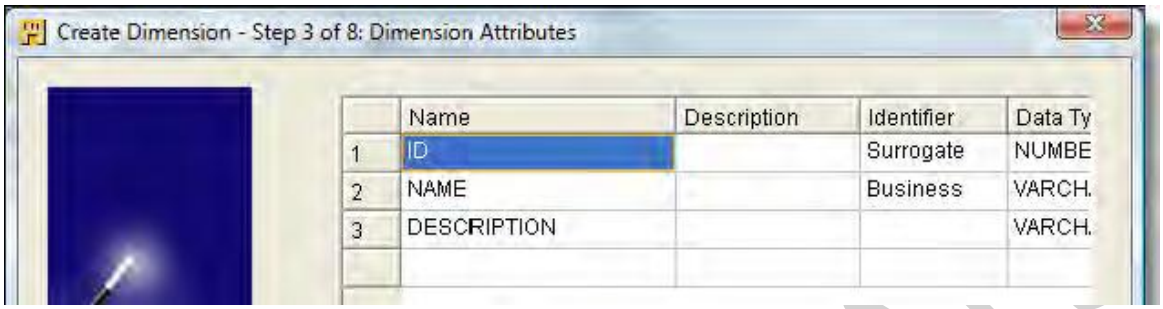
That wizard had a preset number of attributes it defined for us automatically because it knew it was creating a Time dimension. We then had to define the levels where we simply chose from a preset list of levels for the Time dimension. Here we have to explicitly name the levels. This is where we'll have 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.

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:

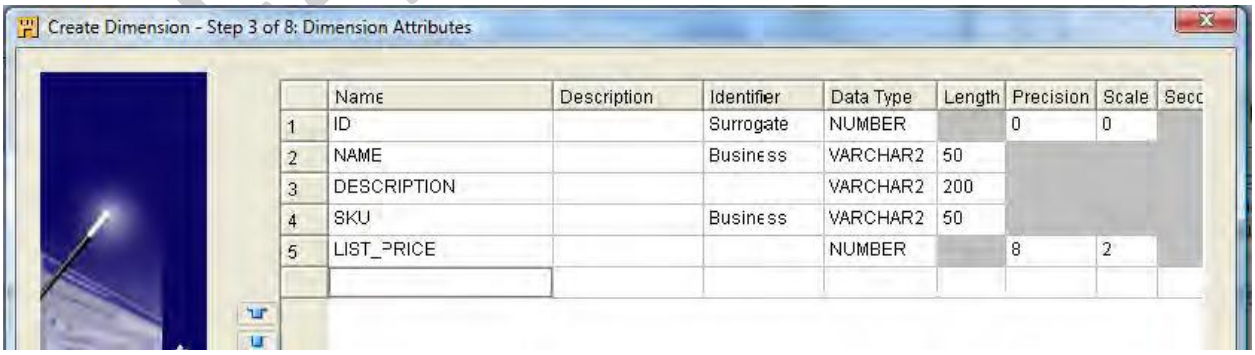


Notice that the wizard has already labeled the **ID** as the **Surrogate Identifier** and the **Name** as the **Business Identifier**, and selected data types for those attributes for us.

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.

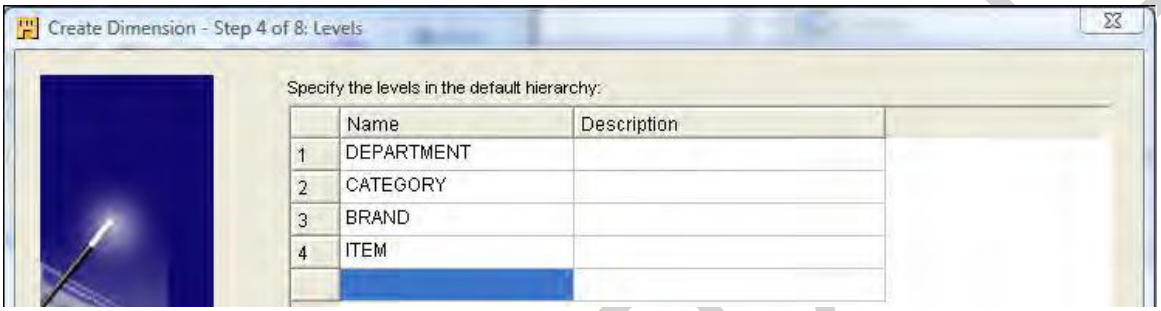
Suppose we make a mistake and enter a value and then decide not to keep it. Then we can delete the row by right-clicking on the row number to the left of the row, and then selecting **Delete** from the pop-up menu.



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.



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.

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.

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.

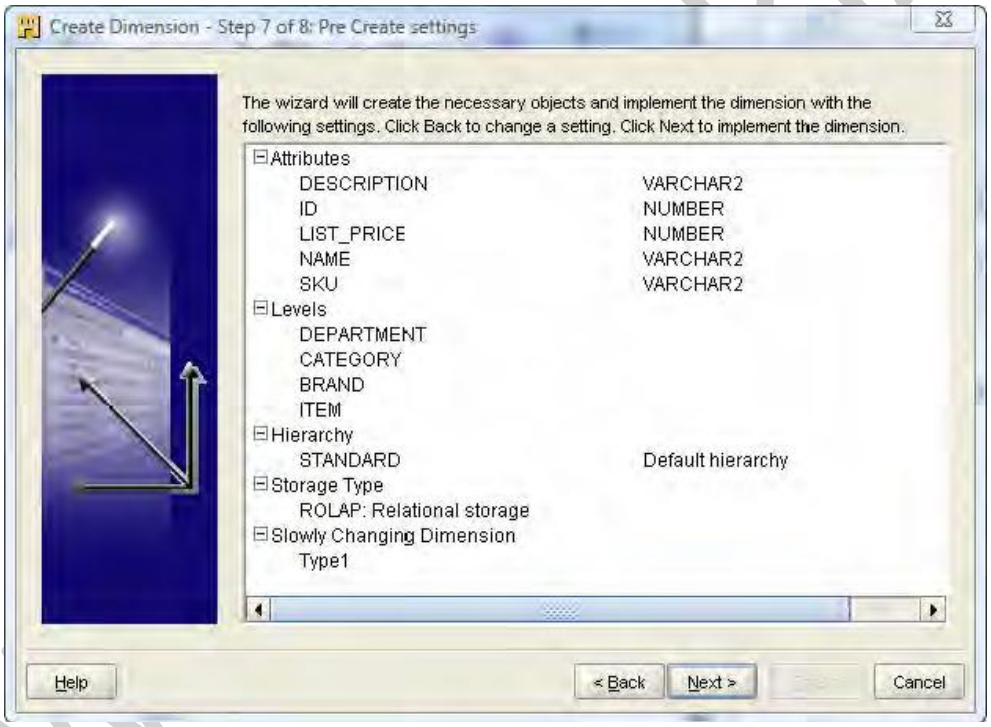
Type 3: Store only the previous value. This means we only care about seeing what the previous value might have been, but don't care what it was before that.

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.

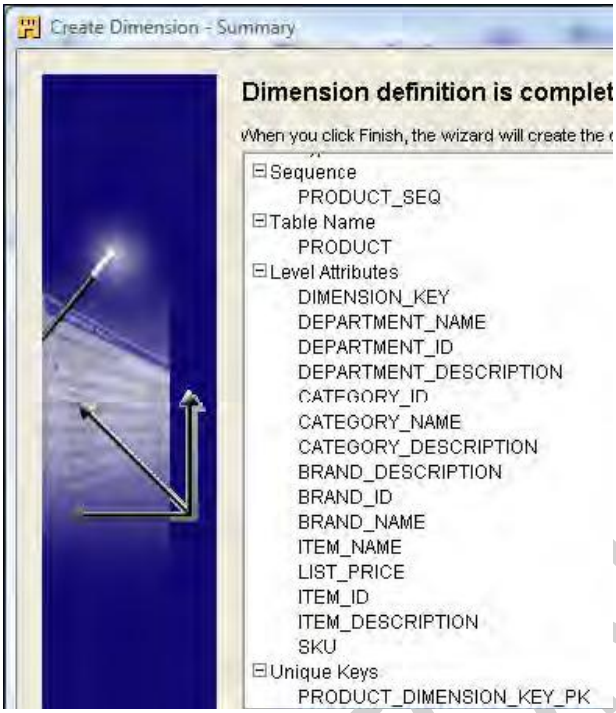
The Type 2 option to maintain a complete history would result in needing additional attributes where we want to maintain historical information. We need attributes designated as **Triggering attributes**. If changed, these attributes will generate a historical record. We also need an **Effective Date 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.

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:



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.



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.

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.

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

Creating the Store dimension with the New Dimension Wizard

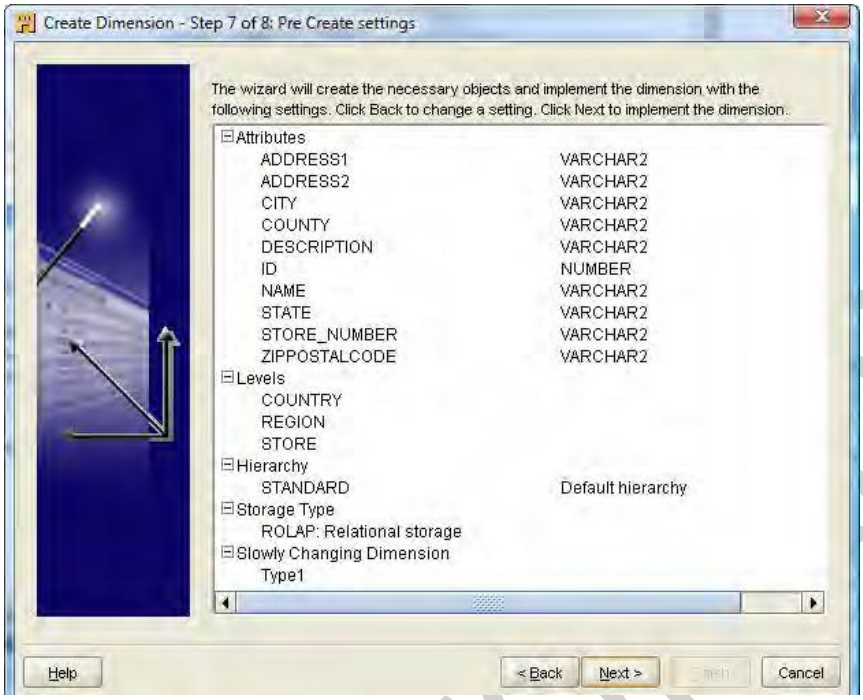
In step 3, where we put in the attributes listed previously, we need to make sure not to forget to specify the surrogate and business identifiers. The surrogate identifier can stay as the default on the `ID`, but we will have to change the business identifier to be the `STORE_NUMBER`, which is a unique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

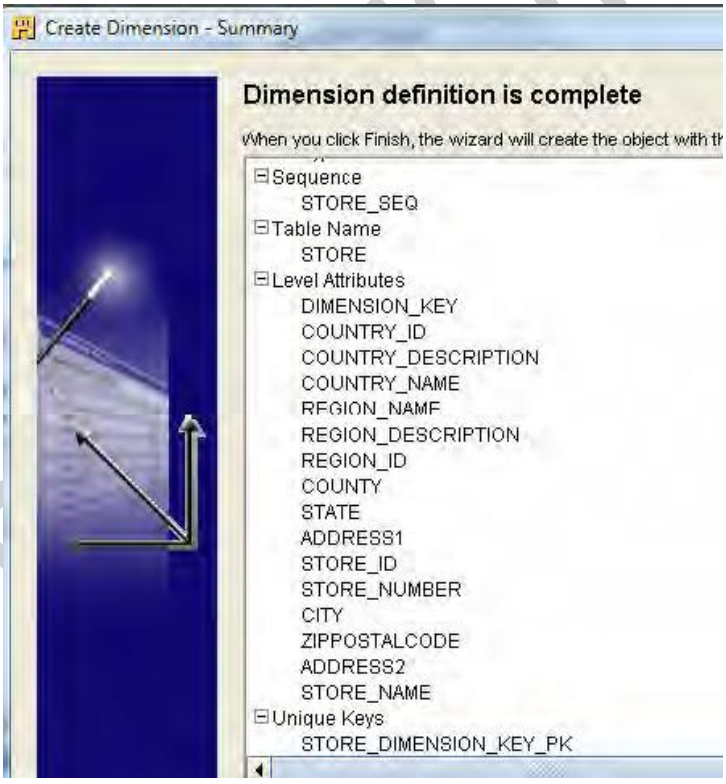
In step 5 where we specify the level attributes, (the above-listed attributes that are applicable to each level) we need to specify all the attributes except `DESCRIPTION` for the `Store` level, and then just `ID`, `NAME`, and `DESCRIPTION` for the `Region` and `Country` levels. This is how we will include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

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.

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.





Creating a cube in OWB

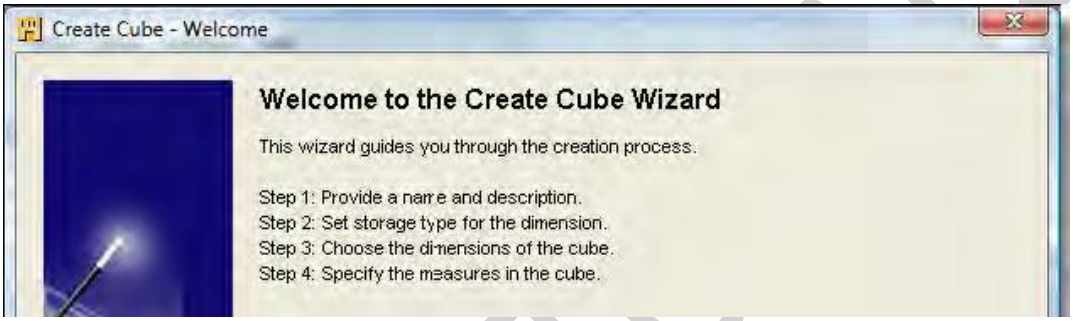
We need to define our cube, which is where our measures will be stored—the facts that users will want to query. We discussed the design of our cube and agreed that we would store two measures, namely

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.

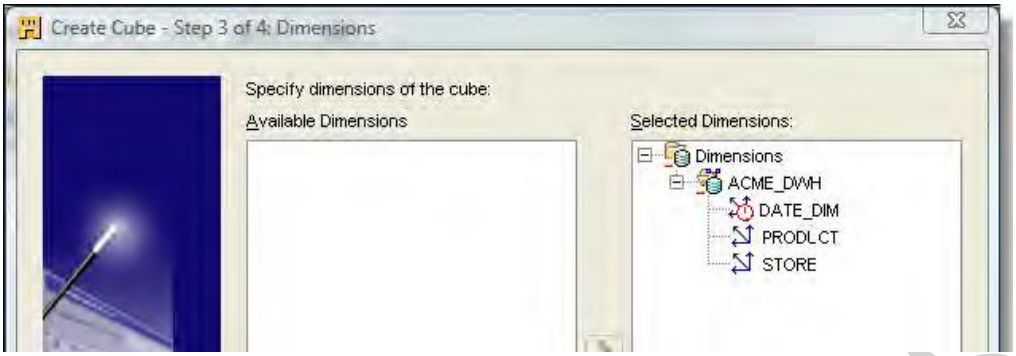


The following are the steps in the creation process:

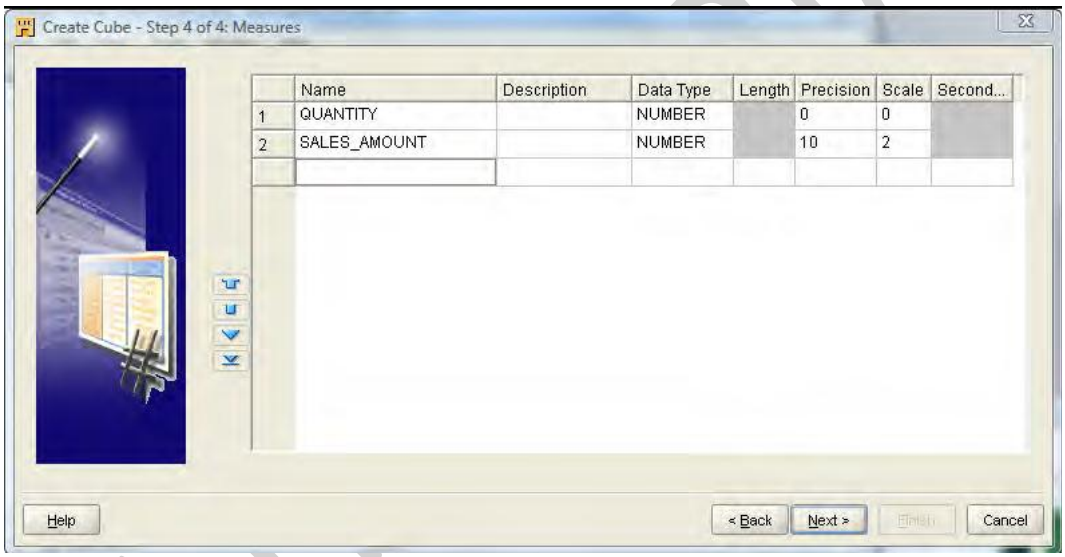
1. We proceed right to the first step where we give our cube a name. As we will be primarily storing sales data, let's call our cube **SALES** and proceed to the next step.

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:

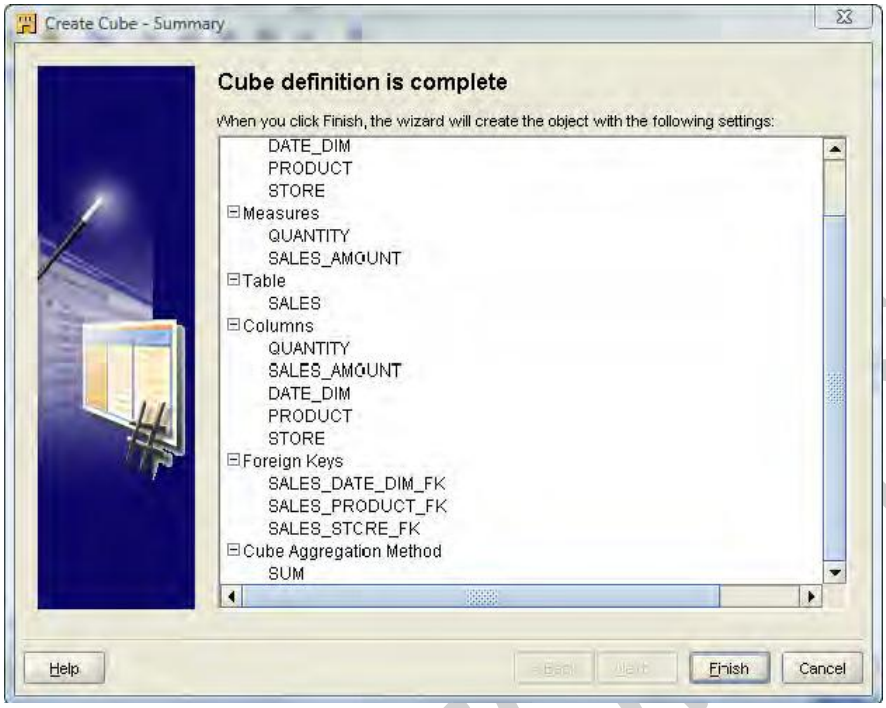


4. Moving on to the last step, we will enter the measures we would like the cube to contain. When we enter QUANTITY for the first measure and SALES_AMOUNT for the second one, we end up with a screen that should look similar to this with the dialog box expanded to show all the columns:



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:



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

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.

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.

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.

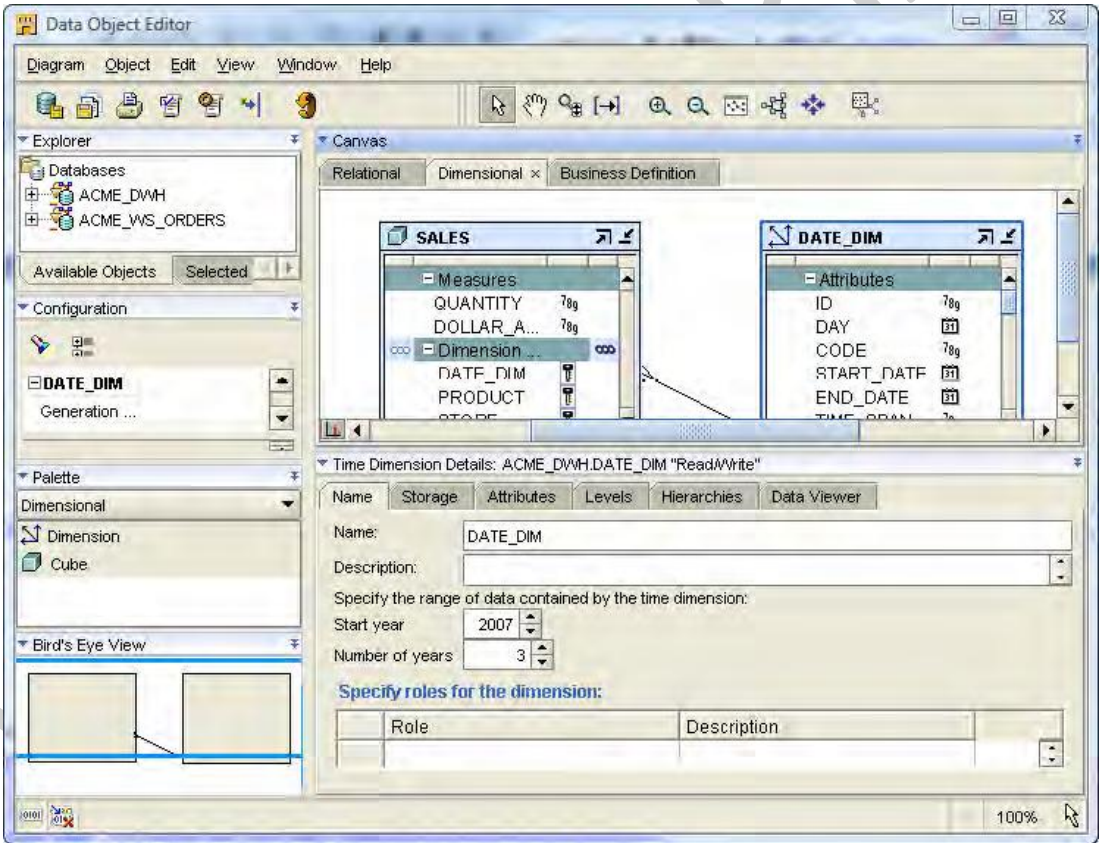
The aggregation the cube will perform for us when we view different levels is one of those behind-the-scenes capabilities we would get with the OLAP feature. When we view the region amounts, they will automatically be summed up from the amounts of the various stores in the region without us having to 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. 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

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 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:



- **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**.

- 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.

- **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.

- **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.

- **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.

1. **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.

2. **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.

3. **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.

4. **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.

5. **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.

6. **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.

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.

Unit 4

4.1 Extract, Transform, and Load Basics

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.

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.

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

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.

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.

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.

Our staging area in this case would be a folder on the file system and the data would be stored in a flat file.

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.

Mappings and operators in OWB

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**.

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.

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.

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.

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.

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.

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.

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`.

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

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.

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.

The operator on the right of the **DATE_INPUTS** operator called **DAY_TABLE_FUNCTION**.

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.

OWB operators

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.

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.

Cube Operator—an operator that represents a cube. This operator will be used to represent that cube in our mapping.

Dimension Operator—an operator that represents previously defined dimensions. 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.

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.

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.

Construct Object—this operator can be used to actually construct an object in our mapping.

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.

- ## Data flow operators

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.

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.

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.

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**.

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.

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.

Unit 4

4.2 Designing and building an ETL mapping

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

Sales: The data elements in the Sales dimensional object are:

- Quantity
- Sales amount

Data: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, Category, and Brand

Store: The data elements in the Store dimensional object are:

- Name
- Number
- Address1
- Address2

- City
- State
- Zip postal code
- Country
- Region

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.

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.**
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'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.

✓ Indexes

The next tab provided in the Data Object Editor is the Indexes tab.

✓ 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.

✓ 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.

Designing our mapping

Now that we have our staging table defined, we are now ready to actually begin designing our mapping.

Review of the Mapping Editor

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.The **Data Object Editor**, the **Mapping Editor** has a blank area named **Mapping** where the **Data Object Editor** called it the **Canvas**.The **Mapping Editor** has an **Explorer** window in common with the **Data Object Editor**

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.

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.

Adding source tables

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. 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.

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.

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'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.

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.

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.

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.An **Aggregator** operator that can be used to aggregate data. 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.

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`.

Add when we click on the **Add** button is an input group. Now we'll click on the **OK** button to close out the **Joiner Editor** dialog box.

Connecting operators to the Joiner

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**.

Defining operator properties for the Joiner

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**. Click on the blank box to the right of the **Join Condition** label.

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.

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.

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.

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.

Adding an Aggregator operator

The Aggregator operator requires that we specify a few things for it to function correctly. 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.

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:

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.

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.

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.

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.

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.

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.

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.
```

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.

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**.

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.

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

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**

Unit.5

5.1 ETL: Transformations and Other Operators

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.

STORE mapping

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.

Adding source and target operators

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.

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

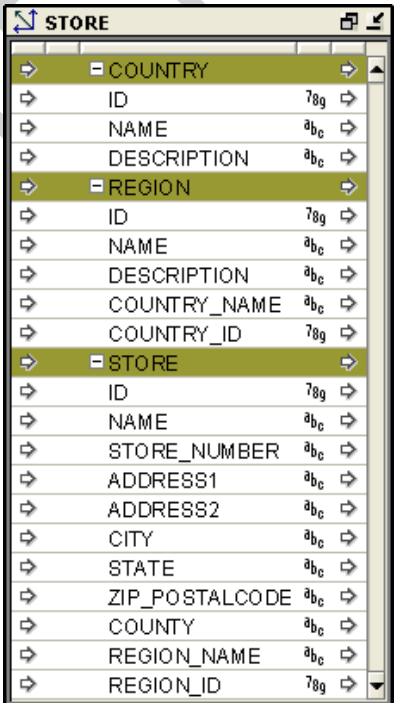
For the REGION level, we'll need data for the following attributes:

- 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.

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
- 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

We'll see how to implement the necessary transformations in the mapping to correct them:

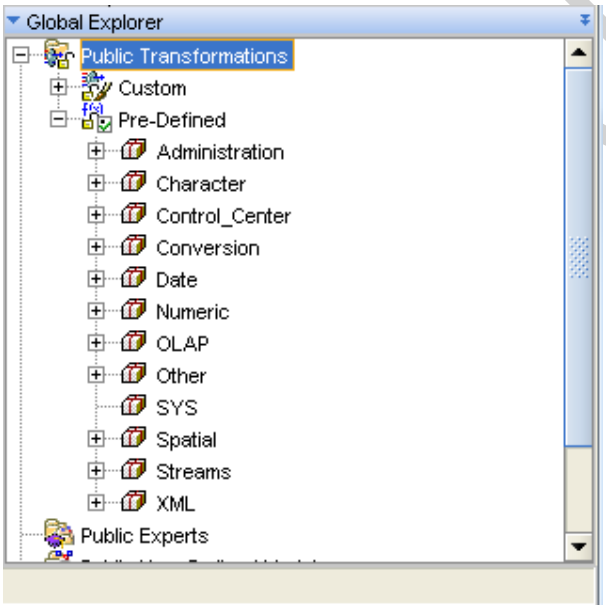
- `STORE_NAME`, `STORE_NUMBER`: We need to trim spaces and change these attributes to uppercase to facilitate queries as they are part of the business identifier
- `STORE_ADDRESS1`, `ADDRESS2`, `CITY`, `STATE`, and `ZIP_POSTALCODE`: We need to trim spaces and change the `STATE` attribute to uppercase
- `STORE_REGION`: We need to trim spaces and change this attribute to uppercase.

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**.

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.

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**.



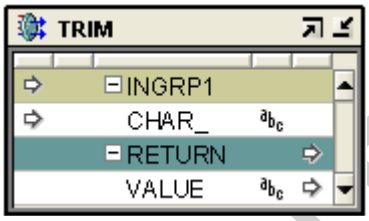
The **Transformations** heading under the **ETL Design Reference** section of the online help table of contents. You can access this by selecting **Help | Table of Contents** from the main menu of the Design Center After dropping the **Transformation Operator** on the mapping, it will pop up a dialog 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.

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.

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.

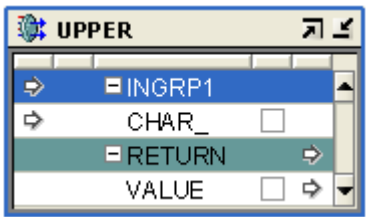


Upper and lowercase issues

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.

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**.

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 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.

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.

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**.

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.

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

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.

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.

Creating an external table

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:

- 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**.
- 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.
- 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.
- 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.

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:
- Right-click on the **Mappings** node, select **New...**, enter **COUNTIES_LOOKUP_MAP** in the name field, and click on the **OK** button.
- In the **Mapping Editor** that pops up, let's drag an **External Table Operator** from the **Palette** window onto the mapping.
- 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.
- 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.
- 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.

- We'll click on **OK** and it will drop a Table Operator onto our mapping with no attributes defined in it.

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:

- 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.
- 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.
- 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.
- 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.

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:

- In the **Design Center**, open the **COUNTIES_LOOKUP** table in the **Data Object Editor** by double-clicking on it under the **Tables** node.
- Click on the **Constraints** tab.
- Click on the **Add Constraint** button.
- Type **PK_COUNTIES_LOOKUP** (or any other naming convention we might choose) in the **Name** column.
- In the **Type** column, click on the drop-down menu and select **Primary Key**.
- Click on the **Local Columns** column, and then click on the **Add Local Column** button.
- Click on the drop-down menu that appears and select the **ID** column.

- Close the **Data Object Editor**.

Retrieving the key to use for a Lookup Operator

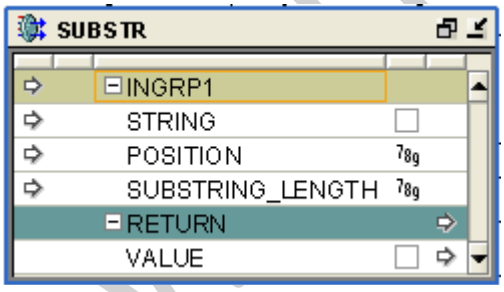
The STORE_NUMBER data element contained in the STORES source table has a code that indicates the county the store is located in for stores in the USA. This is actually a fixed known format, and the positions three through six of the number are actually the code for the location of the store in the county. 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.

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.



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

.

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.

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**

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.

To add an attribute, click on the **Add** button.

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 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.

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.

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**.

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.

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 draga line from the **VALUE** attribute of **SUBSTR** to the **EXPR** input attribute of the **TO_NUMBER** operator.

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.

- 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**.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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 will click on the **Next** button to proceed to the **Summary** screen.

- 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.

- 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.

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.

- 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.

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
- `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.

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

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

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.

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.

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. 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.

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.

One issue is that we didn't create that dimension manually; it was created for us by the Time Dimension Wizard.

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.

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

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.

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.

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

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**

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

The steps to perform for the required expression are:

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.

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.

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.

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.

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**.

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:

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.

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**

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.

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'll now finish entering our expression by typing in the following text to close out the expression:

```
,'YYYYMMDD'))
```

We'll click on the **Validate** button to make sure our expression is correctly entered and our **Expression Builder** window

We'll click on the **OK** button to close the **Expression Builder** and our mapping now validates successfully.

Features and benefits of OWB

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.

Unit .5

5.2.Validating,Generating,

Deploying, and Executing Objects

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.

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.

Validating

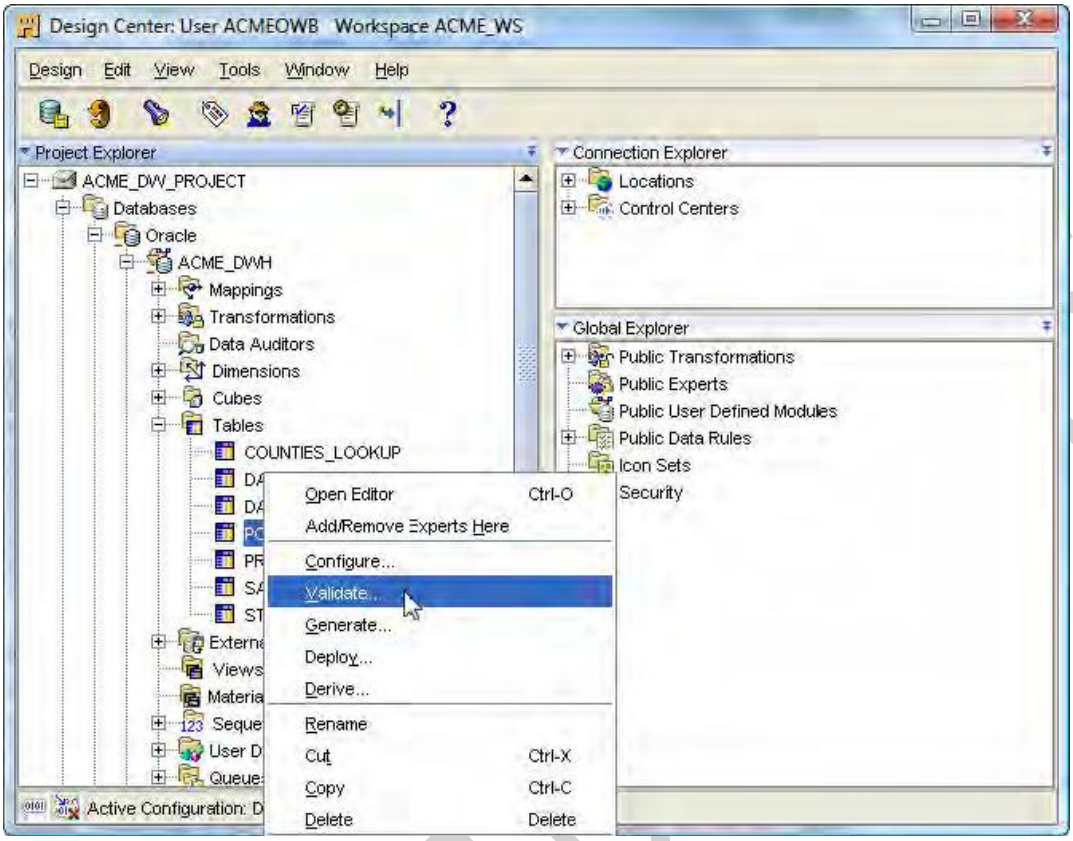
Error checking is what validation is for. The process of validation is all about making sure the objects and mappings we've defined in the Warehouse Builder have no obvious errors in design.

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.

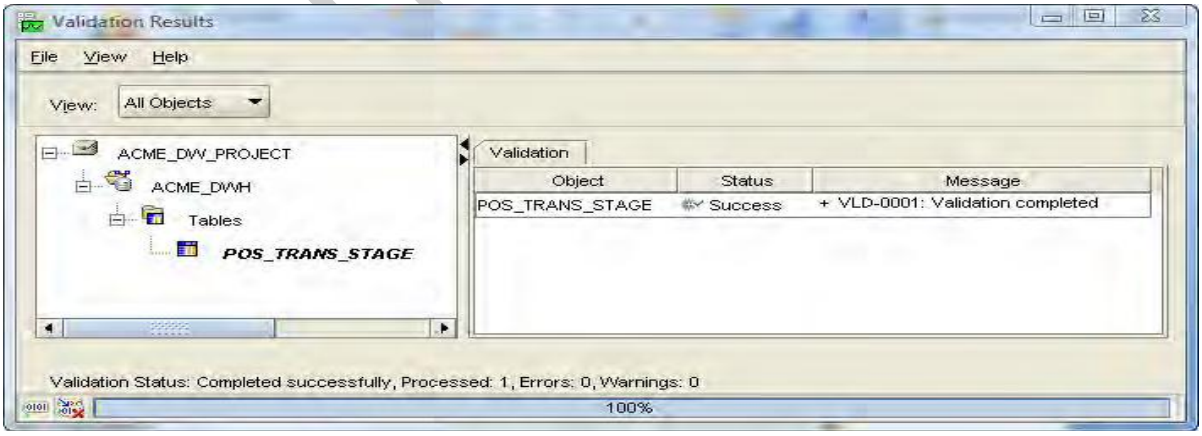
Validating in the Design Center

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:



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 shownnext:



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.

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.

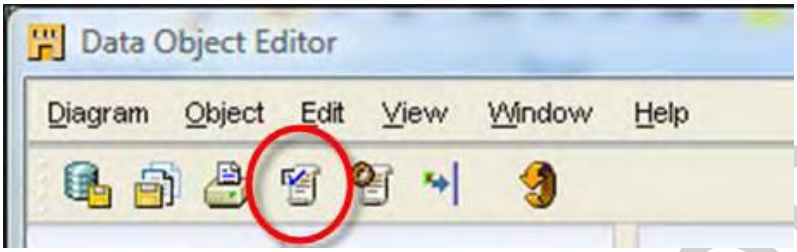
We can close the **Validation Results** dialog box now before moving on to discuss validating from the editors. We can click on **X** in the dialog box window header, or click on the **Close** entry from the **File** main menu, or use the *Ctrl+F4* key combination.

Validating from the editors

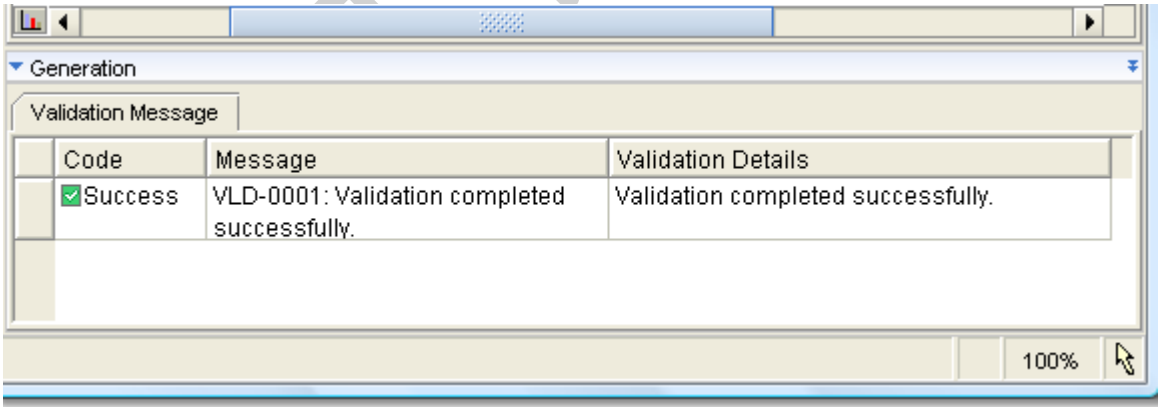
The Data Object Editor and the Mapping Editor both provide for validation from within the editor. We'll talk about the Data Object Editor first and then about the Mapping Editor because there are some subtle differences.

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:



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.

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.

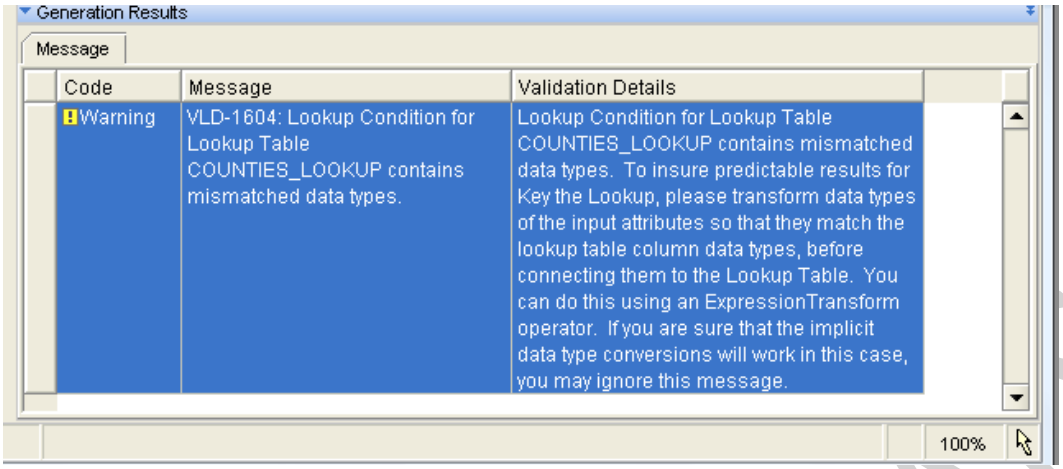
Validating in the Mapping Editor

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.

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.

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.



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.

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.A NUMBER data type is a built-in data type in the Oracle Database.

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.

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.

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.

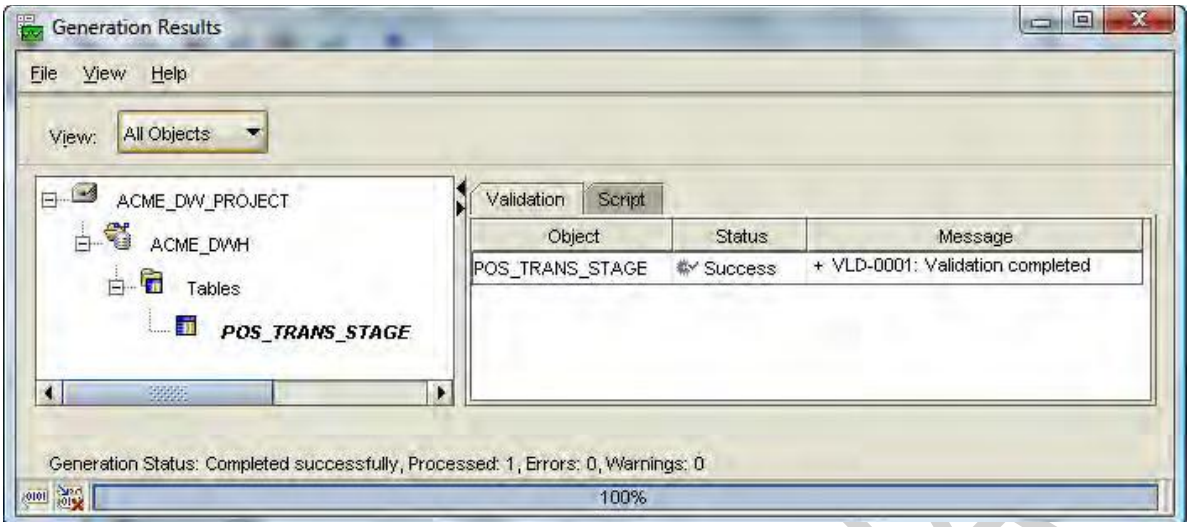
Generating

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.

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.

Generating in the Design Center

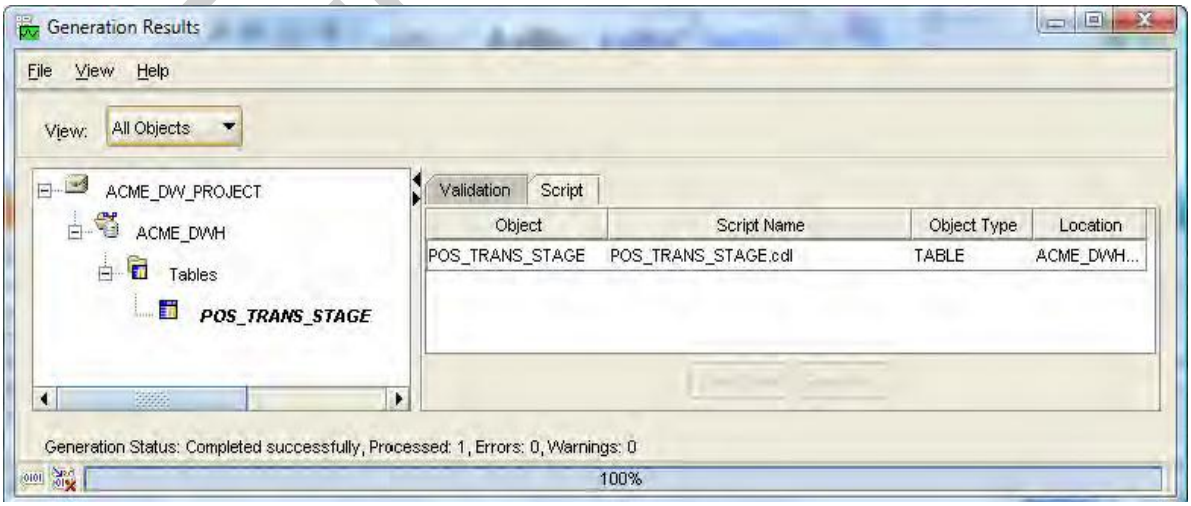
When we generate an object or mapping in the Design Center, we'll get the same pop-up window appearing as we got when we validated, but in this case the window will be labeled **Generation Results** and we'll have some extra information displayed. Let's go to the Design Center now and generate the code 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:



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:

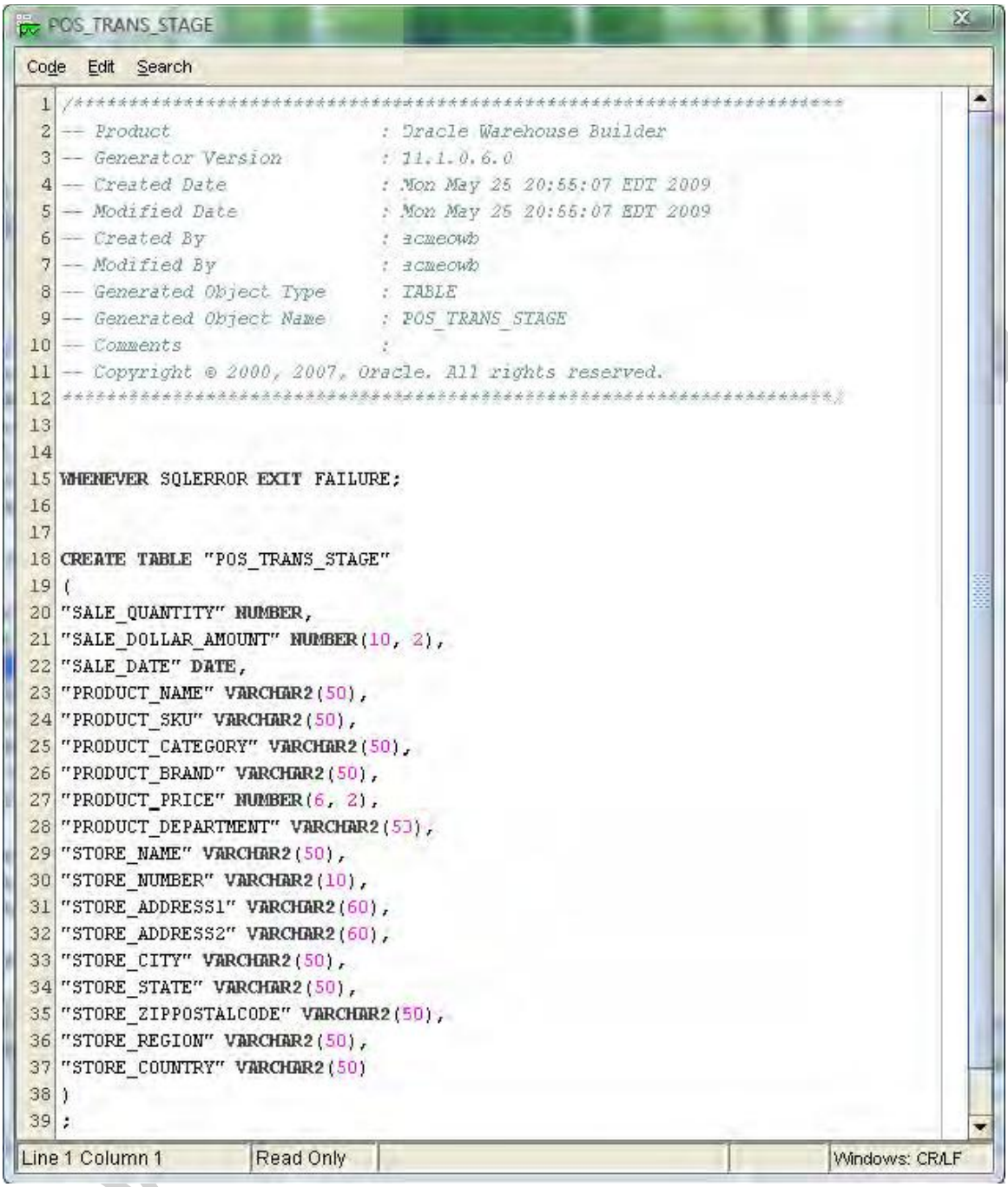


We have two options in this dialog box:

We can view the script

We can save it to the disk

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 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:

Saving this code to a file

Copying portions of the code and pasting them into another window

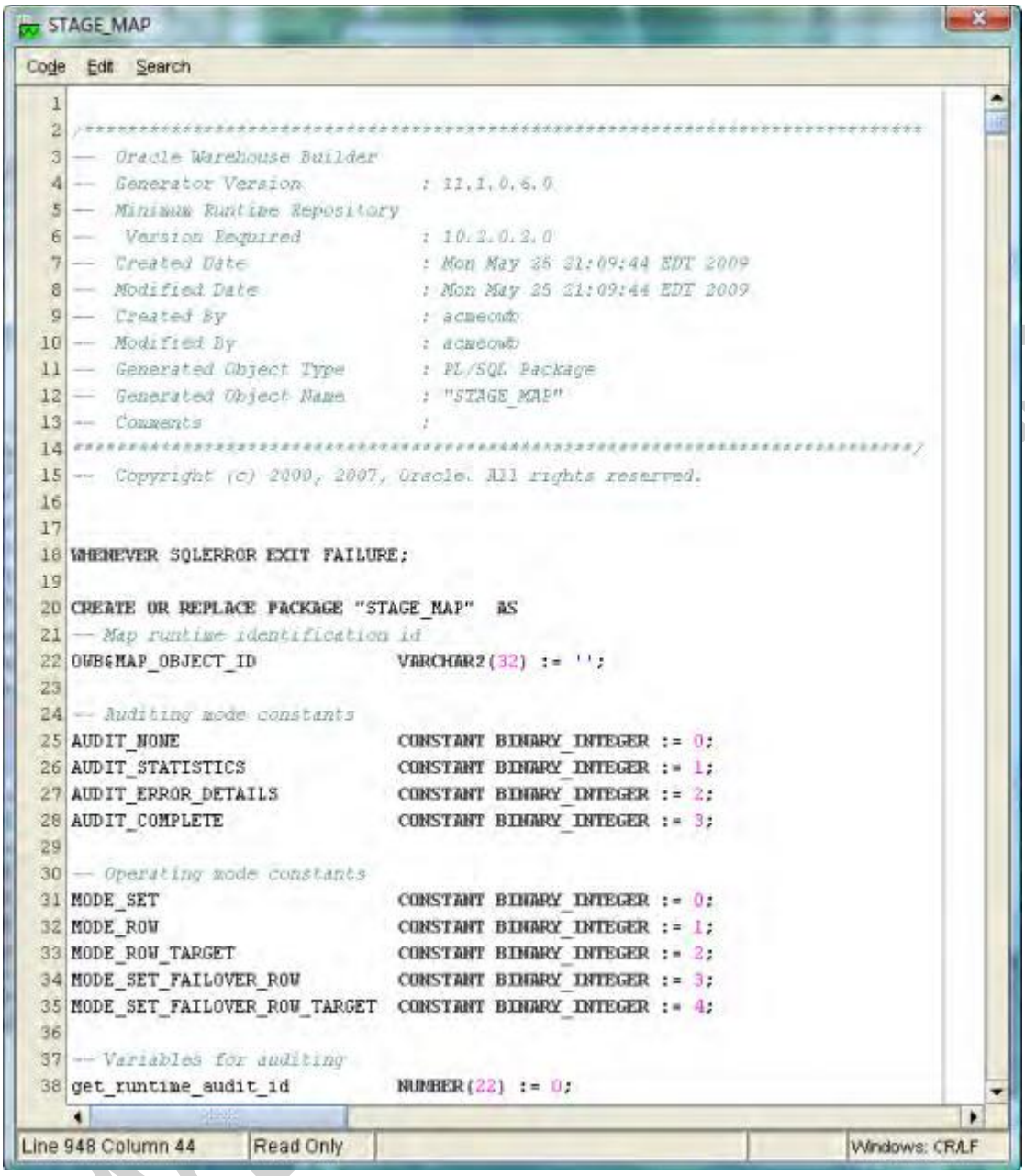
Searching through the code for text strings

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:

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.



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.

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.

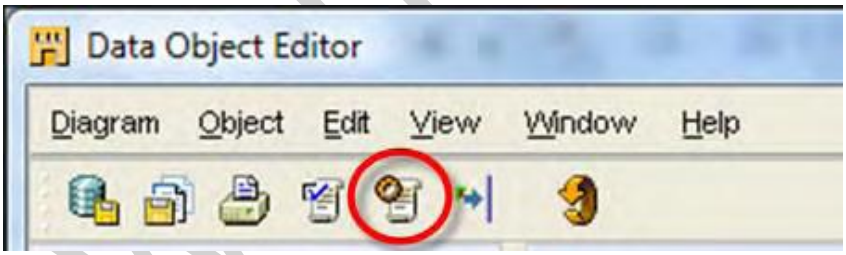
As with validation, the generation function can be run from the editors as well and this is slightly different from running it from the Design Center. So let's take a look at that now after closing out the script window and the generation results dialog box.

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.

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:

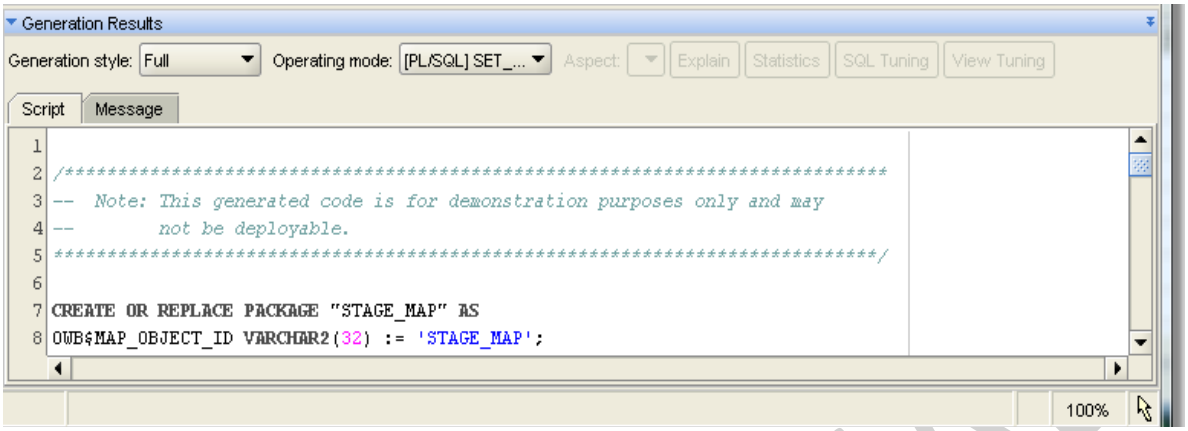
The window also provides us a **Validation Messages** tab, which will display any messages generated as a result of validation. We also have a menu of options that appears on the **Script** tab: **Code**, **Edit**, and **Search** that provide us the same options as on the pop-up window when viewing the script from the Design Center that we saw previously.

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.



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:

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.

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 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.

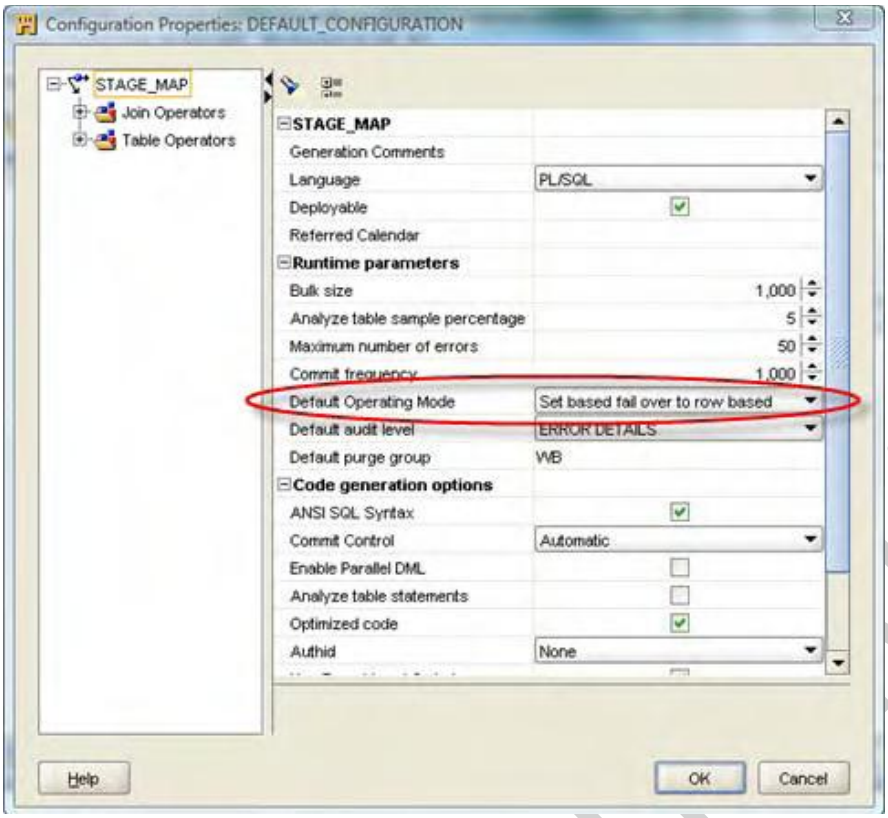
In row-based mode, the Warehouse Builder generates code to process the data row by row. It uses a combination of SQL Cursors and PL/SQL code. It does not provide as big a performance benefit as the set-based mode, but we gain much greater auditing capability of the execution results. There are also additional 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.

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**.

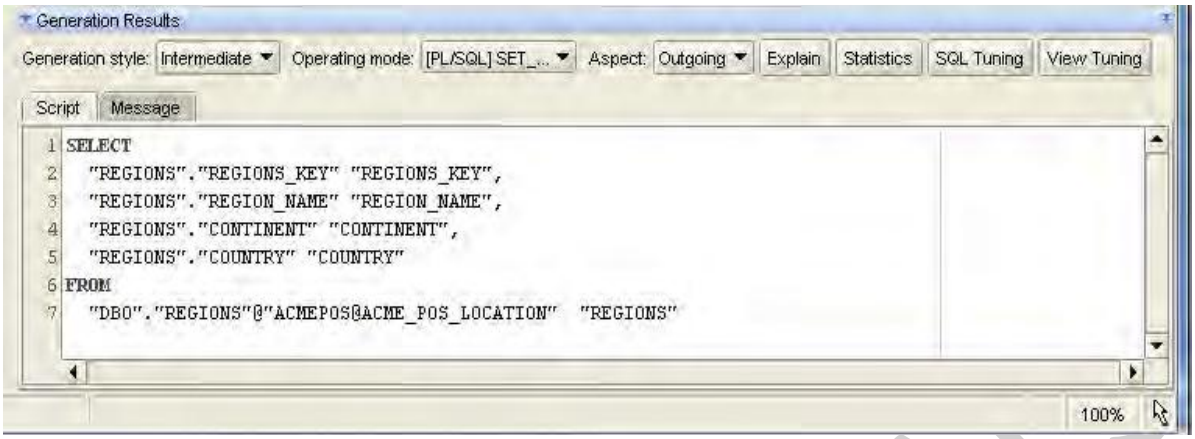


Selecting the generation style

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:



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`.

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.

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.

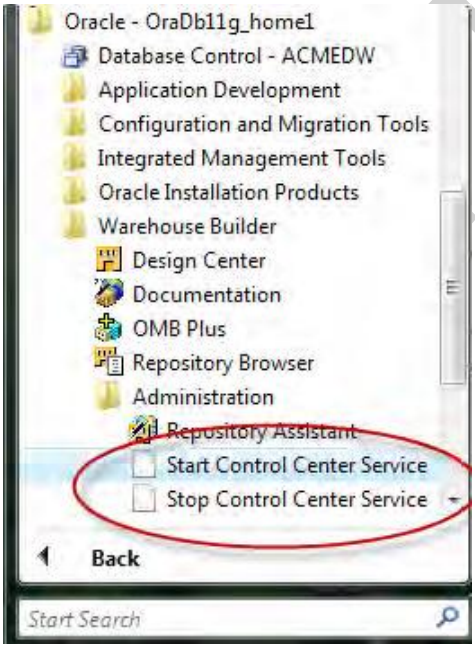
Deploying

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 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.

The Control Center Service

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:



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.

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

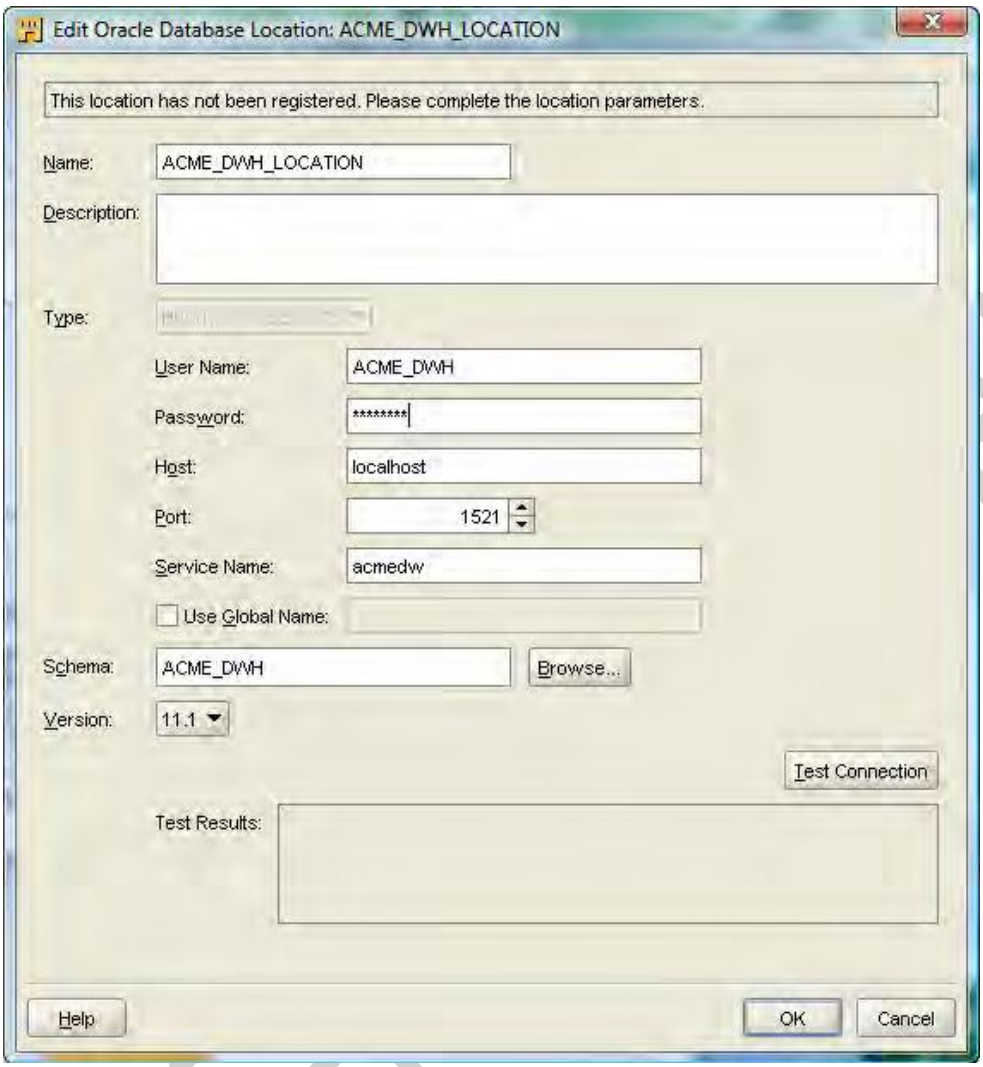
right-click on it and select **Deploy** from the pop-up menu.

If 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

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.

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:



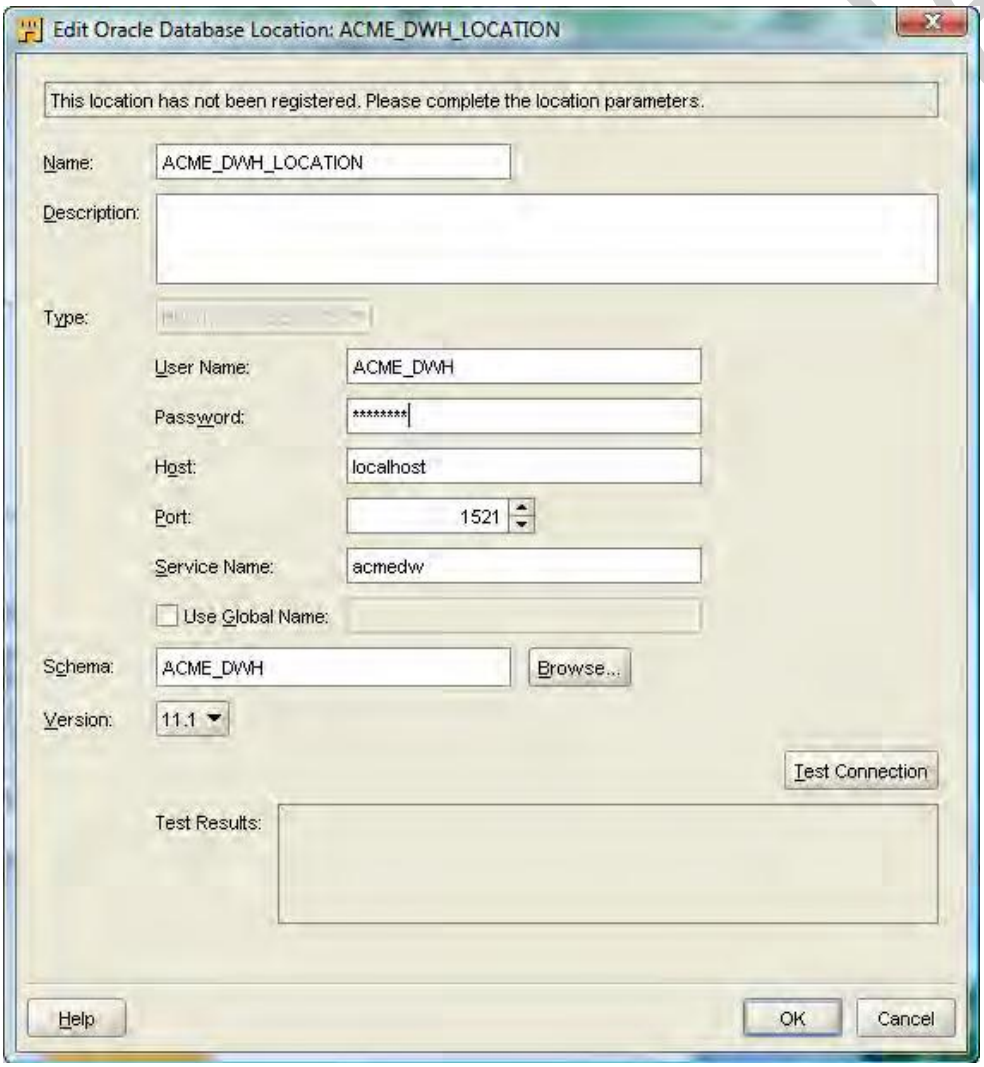
The act of registering a location is to associate all the previous information with a location defined in the Warehouse Builder so that the Control Center Service knows how to find the location. The connection details in the previous dialog box are what it uses to connect to the location. Simply provide 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 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:

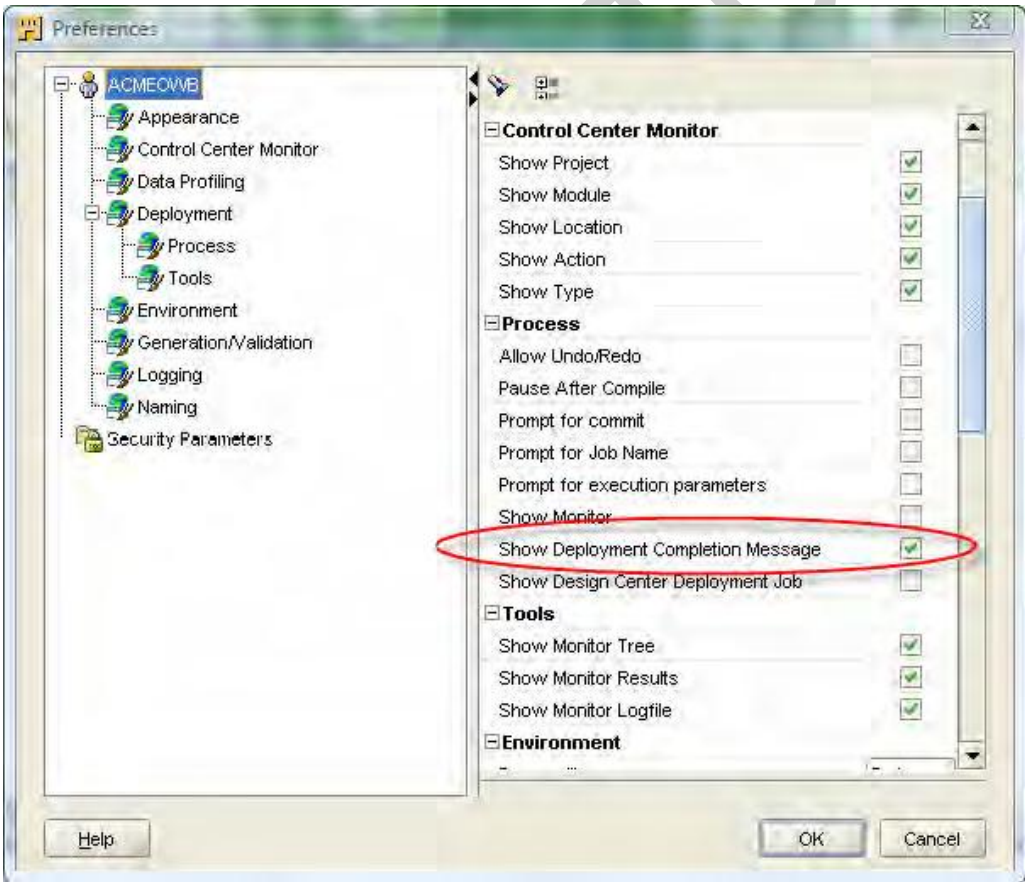
If we get this pop-up window when we click on the **OK** button, we'll get another pop-up window prompting for connection information for the Control Center Service where we can provide connection information. That dialog box looks like the following:



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 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:



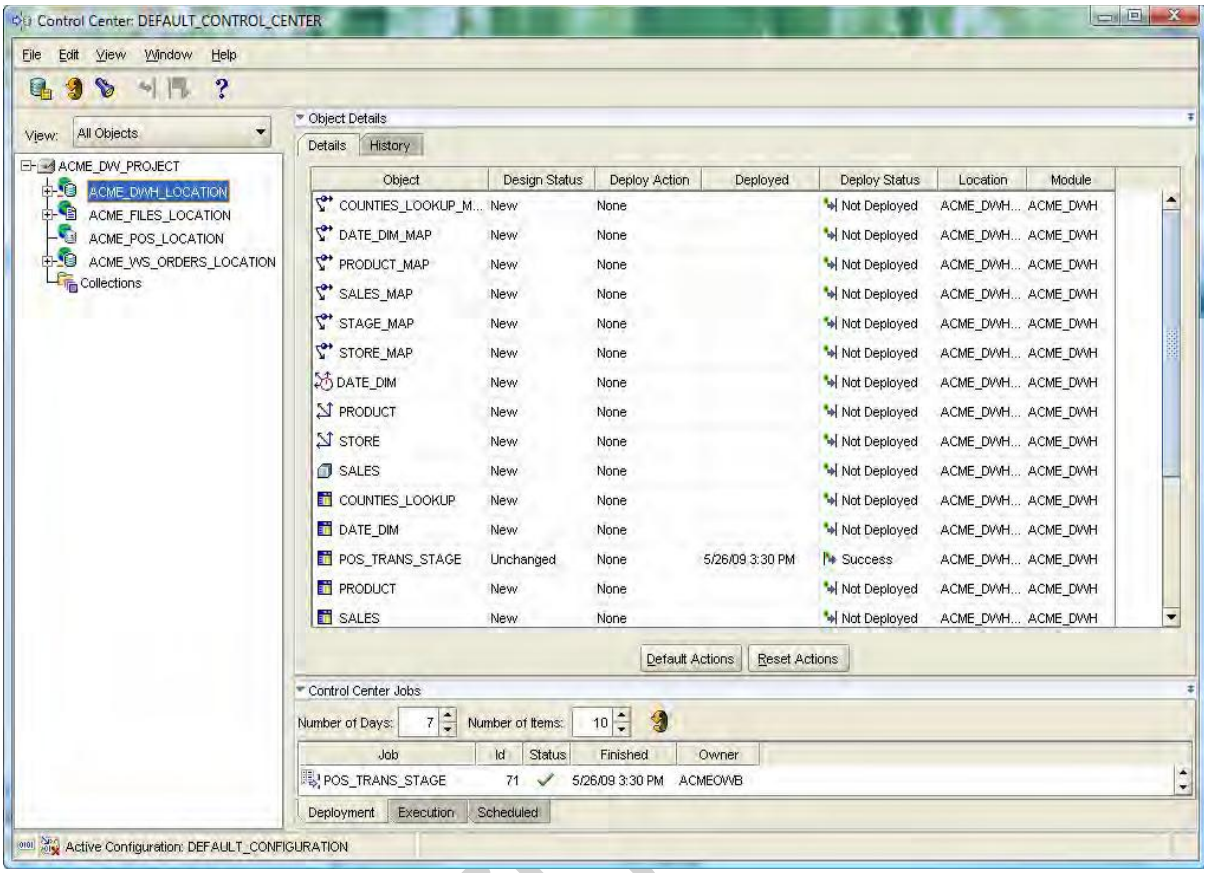
It will show us the completion message after this option has been checked, which will look similar to the following:

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.

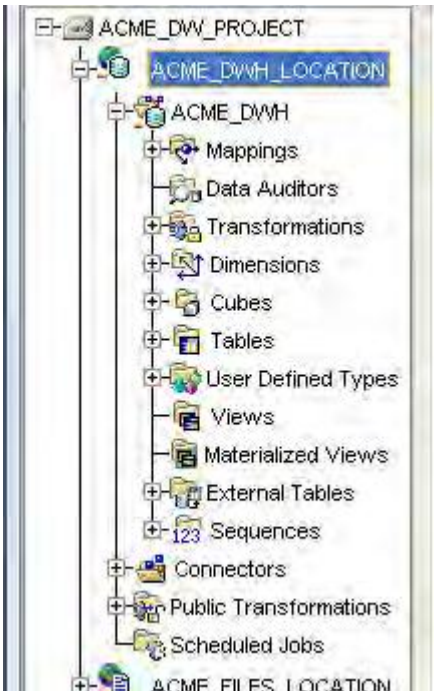
We launch the Control Center Manager from the **Tools** menu of the **Design Center** main menu. We click on the very first menu entry, which says **Control Center Manager**. This will open up a new window to run the Control Center Manager, which will look similar to the following:



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.



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.

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

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

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.

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

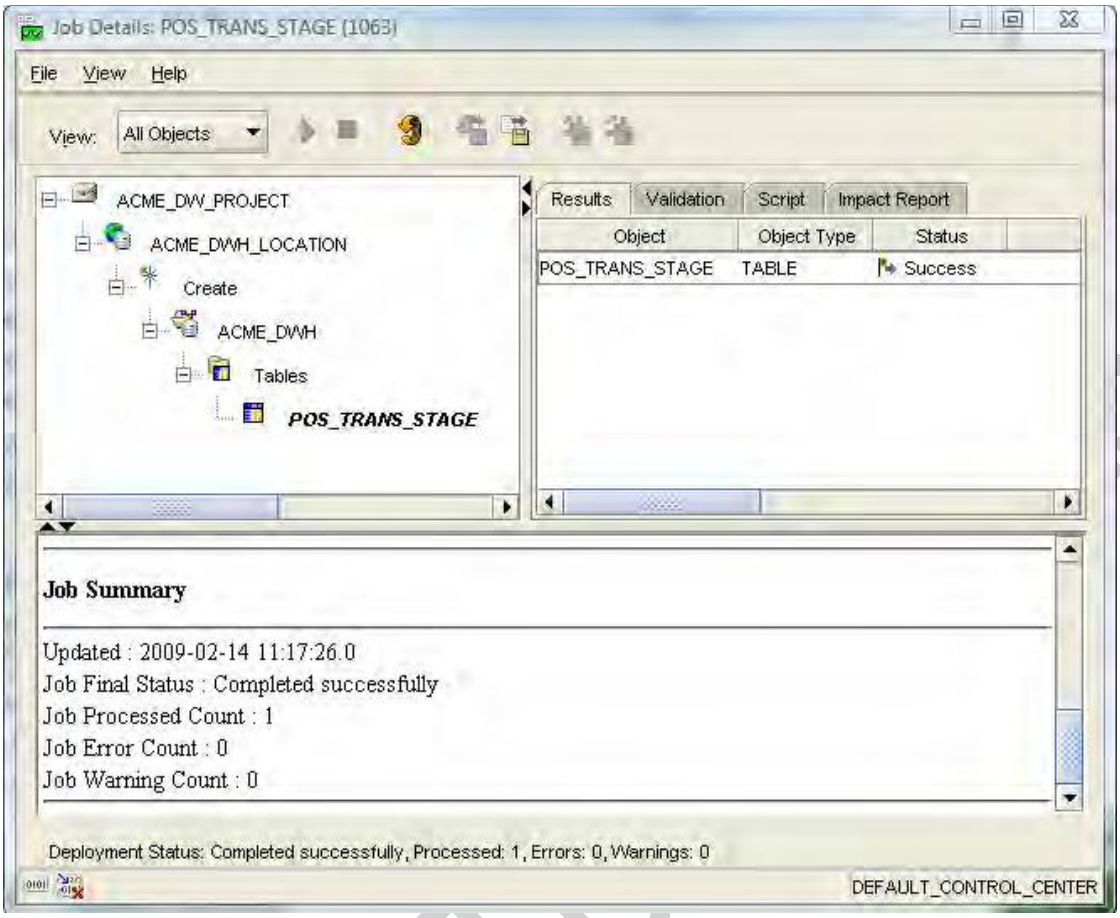
determined by the current design and deployment status. For example, a mapping that has not been deployed yet has a default status of **Create**. A table that was previously deployed but just changed will have a default action of **Upgrade**. The **Default Actions** button will change the displayed **Deploy 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**.

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 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:



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	

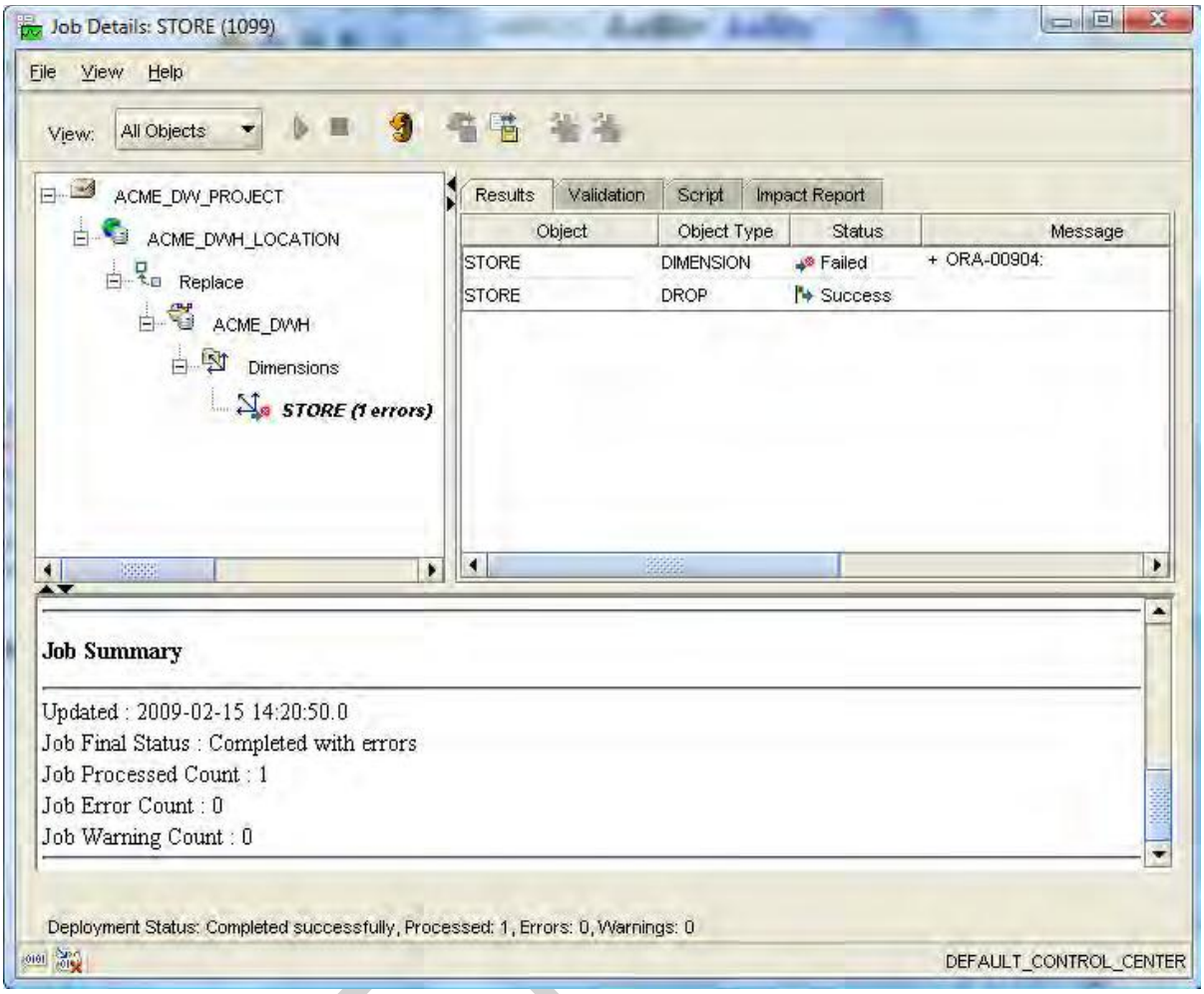
Job Summary

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

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.

The following example will illustrate an error encountered when deploying a dimension, which in this case is the `STORE` dimension:



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 identif:

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.

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 identif:

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.

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.

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.

Executing

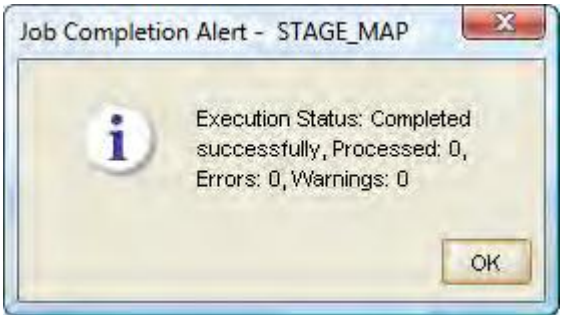
Now we have our staging table deployed to the target database, the `POS_TRANS_ STAGE` table, and have successfully deployed the mapping to load that table from our `STAGE_MAP` source database. This means we now have enough of our target database deployed to be able to execute the `STAGE_MAP` mapping 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.

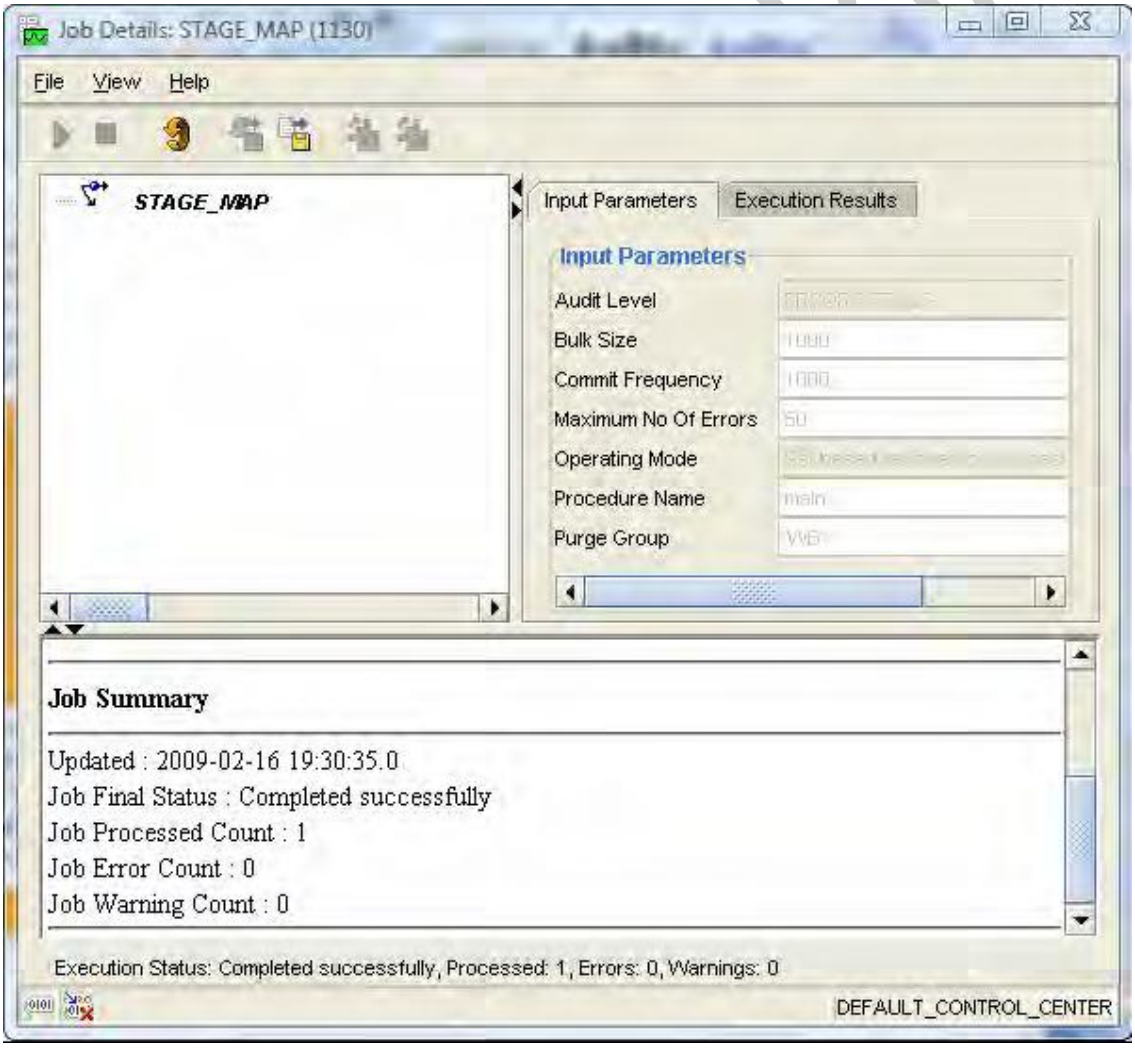
To execute a mapping we might think to look for a menu entry that says **Execute**, but we will not find it. We need to select the menu entry that says **Start** to start the code running. This menu entry is available from the pop-up menu by right-clicking on an item in the tree view, and from the **File** menu when an item is selected in the tree view.

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.

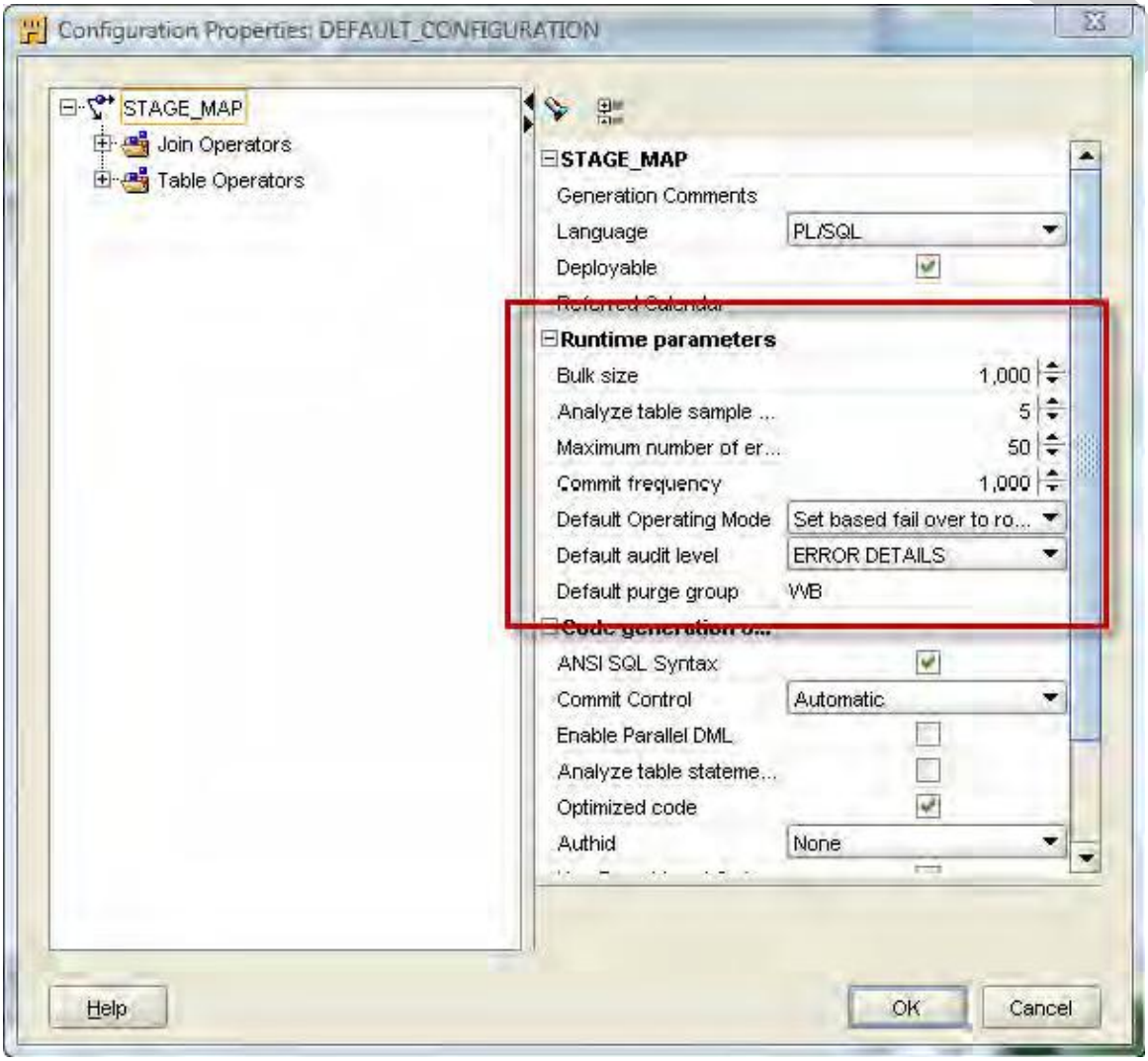


The important thing to notice about this dialog box is the success or failure message. The counts (at least for the processed count) are not accurate. This is a minor bug, as it did indeed process this mapping. This is verifiable by double-clicking on the status for our job in the **Control Center Jobs** window to display the details about this execution.



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**.

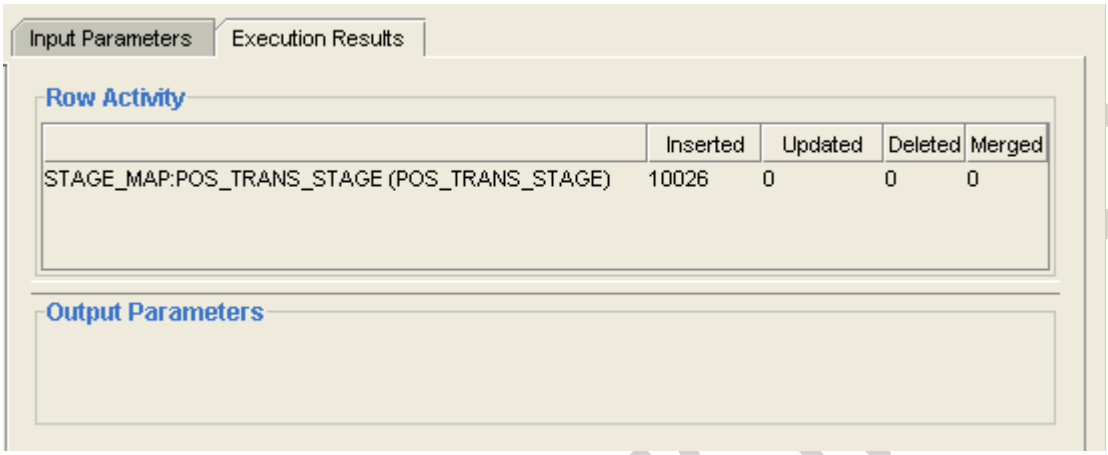
The input parameters are configuration options for running the mapping that involve the operating mode among others. We discussed the operating mode previously when talking about generating code and viewing the code for the various operating modes.



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:



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.

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:

- 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.

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

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.

Unit 6

6.1 Extra Features

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. 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.

Metadata change management:

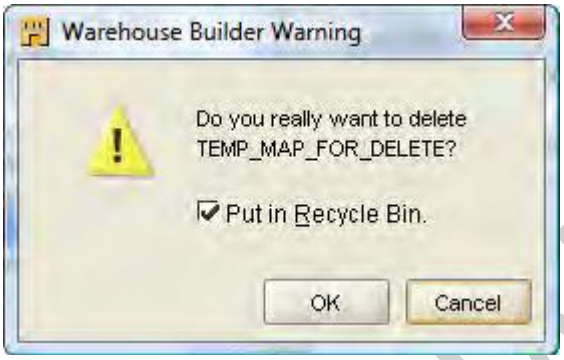
Metadata change management includes keeping a track of different versions of an object or mapping as we make changes to it, and comparing objects to see what has changed. It is always a good idea to save a working copy of objects and mappings when they are complete and function correctly. That way, if 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.

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`.
- 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**.
- 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.

- 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.

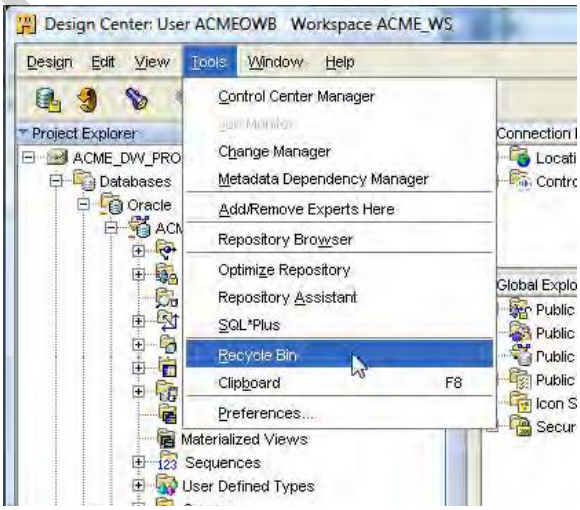


- The option to delete the object and move it to the recycle bin where it would still be accessible later if needed, or just delete the object entirely so that it is never to be seen nor heard from again.

- 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.

- The **Recycle Bin** is accessible from the main menu of the **Design Center** under the **Tools** menu entry as shown next:



- the Recycle Bin window will pop up as shown in the following screenshot:

- 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.

- The Recycle Bin allows us to keep versions of deleted objects. Close the **Recycle Bin** by clicking on the **OK** button.

Cut, copy, and paste:

- 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.

- 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**.

- 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.

- leave the other options set to their default and click on the **Next** button.

- click on the **Next** button and then on **Finish** to create the empty database module.

- 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:
    1. Save our work that we have done so far
    2. Revert back to the previously saved version of our project
    3. Cancel the switching of projects and go back to the project

Click on the **Save** button. That will save the project we just created, close it, and open our original project.

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.

The POS_TRANS_STAGE table on the clipboard, we can paste it into the other project we just created.

- 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 **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.

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.

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.

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.

- Click on the **Next** button to move to step 1 of the wizard where we'll give our snapshot a name. 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. We can click on the **Help** button on this screen to get a detailed description of the two options. 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.

- 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 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.**

- 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.

- Click on **Next** and get the summary display showing us what options we choose.

- 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.

- 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.

    1. **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.

    2. **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.

    3. **Convert to Signature**: This option will convert a full snapshot to a signature snapshot.

    4. **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.

5. **Compare**: This option will let us compare two snapshots to each other to see what the differences are.

- 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.
- 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.
- 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.

Metadata Loader (MDL) exports and imports:

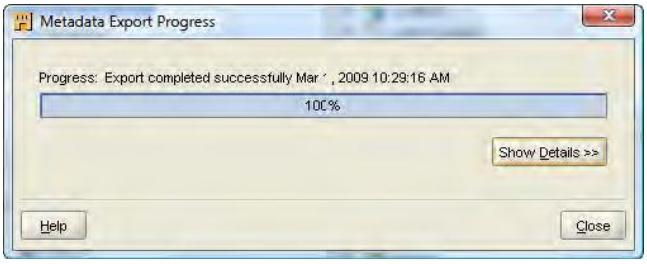
One final change management related tool for managing our metadata that we'll look at in the Warehouse Builder is the ability to export workspace objects and save them to a file using the **Metadata Loader (MDL)** facility.

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.

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.

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.

Accept the default file names and locations, and click on the **Export** button.



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.

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.

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:

1. 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.
2. 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.

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.

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.

Synchronizing:

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 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.

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.

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.

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.

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

Changes to dimensional objects and auto-binding:

With Auto Bind, we can have the Warehouse Builder automatically create the table for us with all the dimension attributes properly bound.

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.

On the **Attributes** tab, scroll down to the end and enter a new attribute called `AREA_SIZE`.

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

6.2 Data Warehousing and OLAP

Defining OLAP:

The term OLAP was first coined by Dr E F Codd, the inventor of the relational model, to describe a genre of software that is used for analyzing business data in a top down hierarchical fashion. It is significant to note that in his white paper, where he coined the term OLAP, Dr. Codd cautioned that "relational databases were never intended to provide the very powerful functions for data synthesis, analysis and consolidation that are being defined as multidimensional analysis" . OLAP tools present data in a multidimensional fashion that non-technical users find intuitive. They are able to navigate through their business data in a simple way, gaining insights which they simply have not been able to gain using previous generation approaches ranging from printed reports to report writers to query tools.

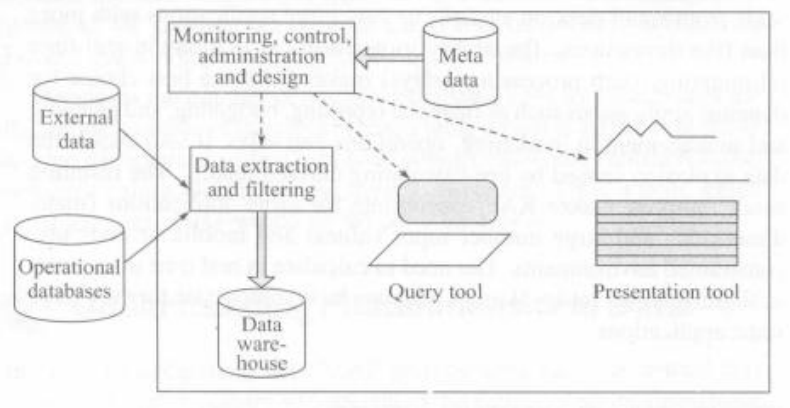
The Value Of Multidimensional Data:

In order to understand the significance of OLAP, it is critical to understand the multidimensional nature of so much of today's business data.

The key thing that matters to him is sales by product. But, for each product, he wants to know unit sales, dollar sales, discounts allowed, and perhaps, certain other key statistics. He wants to know this information on a daily basis, by region, by salesperson, and by channel. What we have identified is a six dimensional model. The dimensions are accounts (often also called variables or measures), products, time, channel, region and salesperson. One of the key features of OLAP technology is that the user is able to navigate through the data in any way that makes sense, without knowing in advance what the navigation route might be. Most organizations plan either from the top down or the bottom up. If the planning is done from the top down the impact of the plan must be projected down to the responsibility center level in order that managers may be held accountable for their plans. If the planning is done from the bottom up then the plans must be consolidated so that top management can agree to the overall plan. In practice, most organizations take a hybrid approach where top management sets some guidelines which the budget center managers use to build their detailed plans. The consolidated plan is then reviewed by top management to refine their guidelines, resulting in a further round of planning at the detailed level. This process may go on for successive iterations. This process is also multidimensional. The dimensions include accounts, responsibility centers, time, versions (actual, plan, revised plan). Additional dimensions commonly included are product and customer type. OLAP tools should be capable of embedding this type of complex business logic in the multidimensional model and be capable of responding to changing assumptions in real time.

OLAP Terminologies:

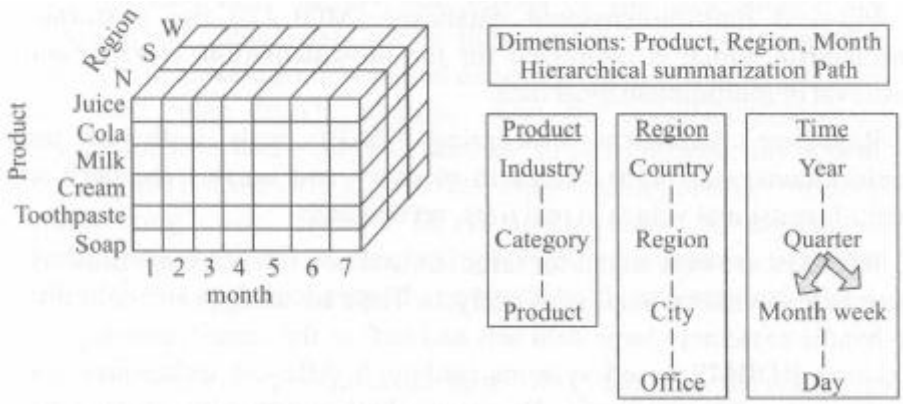
Basic OLAP System Architecture



Physical multidimensional databases (MOLAP) use a storage mechanism which is optimized for the pre-calculation, storage, and retrieval of multidimensional data.Real-time Analytical Processing (RAP) deals with all the multidimensional input values in memory and creates the derived multidimensional values in real time, on demand.ROLAPs are best suited for large, transaction intensive applications such as high volume retail sales analysis. Their advantages are the ability to handle extremely large data sets and having the same technology as existing RDBMS based systems (although different techniques and optimization are required). However, their complexity, storage vs.calculation orientation, cost and performance constraints limit the range of applications for which they are suited. For these reasons, they are not often used for budgeting or business and financial applications. MOLAPs are best suited for medium sized, static, and mostly static applications, which demand nothing less than sub-second data retrieval.

RAP is best suited for dynamic applications, for environments that should support a mobile workforce, and for environments that need to scale from small desktop systems to very large applications with more than five dimensions.

online multidimentional cube



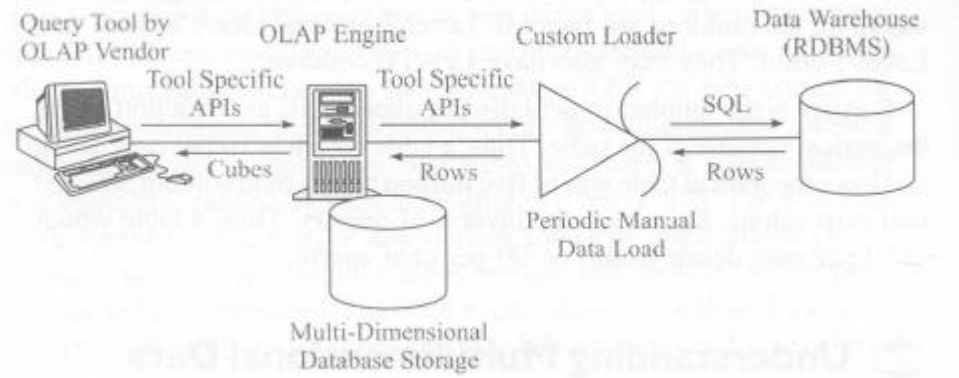
Physical multidimensional databases (MOLAP) use a storage mechanism which is optimized for the pre-calculation, storage, and retrieval of multidimensional data. Real-time Analytical Processing (RAP) deals with all the multidimensional input values in memory and creates the derived multidimensional values in real time, on demand. ROLAPs are best suited for large, transaction intensive applications such as high volume retail sales analysis. Their advantages are the ability to handle extremely large data sets and having the same technology as existing RDBMS based systems (although different techniques and optimization are required). However, their complexity, storage vs.calculation orientation, cost and performance constraints limit the range of applications for which they are suited. For these reasons, they are not often used for budgeting or business and financial applications. MOLAPs are best suited for medium sized, static, and mostly static applications, which demand nothing less than sub-second data retrieval. Examples of applications in this category are the analysis of historical sales and financial information. However, since their batch precalculations can take a long time, they are not optimum for dynamic applications where a result from new or updated data is required. Their batch pre-calculation approach may also make them unsuitable for large, very sparse applications with more than five dimensions as the data explosions can be unmanageable.

Understanding Multidimensional Data:

In order to understand the OLAP architectures that can deliver this value, it is first necessary to understand the nature of multidimensional data. Multidimensional data is almost never 100 per cent dense. Of all the theoretical cells in the database, typically, only a small percentage is populated. Consider a real world example of theoretical versus actual hypercube sizes. Even though a medium-sized table could contain a theoretical 32 million cells generally less than 1,000,000 were actually populated. Although this sounds like a low' number, this is a real example and is fairly typical of financial data of this form.

Multidimensional Architectures:

multidimentional database architecture



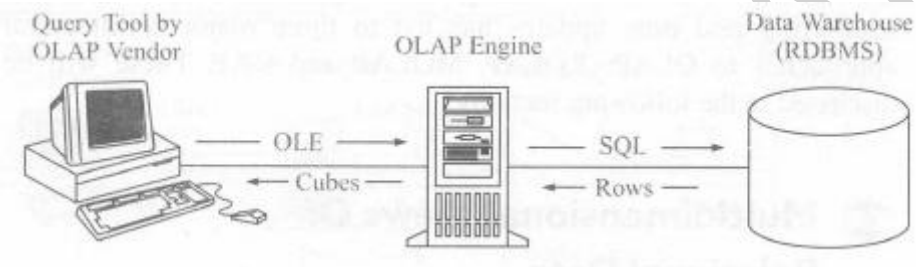
The simplest request is a two dimensional slice of data from the n-dimensional hypercube. The objective is to retrieve the data equally fast, regardless of the requested dimensions. In practice, such simple slices are rare; more typically, the requested data is a compound slice where two or more dimensions are nested as rows or columns. Put another way, the goal is to provide linear response time regardless of where the data is being retrieved from in the hypercube.

A second role of the server is to provide calculated results. By far the most common calculation is, in fact, aggregation, but more complex calculations such as ratios and allocations are also required. In fact the design goal should be to offer a complete algebraic ability where any cell in the hypercube can be derived from any others, using all standard business and statistical functions including conditional logic. Most OLAP servers in fact, achieve fast response to computed results by computing them in advance. This technique can be very effective but will not work well where the size of the fully calculated model is thousands of times greater than the volume of the input data. This may sound unlikely but, in fact this can easily happen, particularly where the number of dimensions is large and the hierarchies in each dimension are deep. This pre-calculation technique is also not effective in the case where the input data is being updated in real time such as in interactive budgeting or financial reporting applications. This is because analysts want to see the effect of changes immediately. Forcing a batch mode recalculation of some or all of the hypercube is very disruptive to the process.

Multidimensional Views Of Relational Data

Some vendors take the view that all data should be stored in relational databases. They provide a multidimensional view of this data. For this, all of the relational OLAP vendors store the data in a special way known as a star or snowflake schema. The most common form of these stores the data values in a de-normalized table known as the fact table. One dimension is selected as the fact dimension and this dimension forms the columns of the fact table. The other dimensions are stored in additional tables with the hierarchy defined by child-parent columns. The dimension tables are then relationally joined with

the fact table to allow multidimensional queries. The data is retrieved from the relational database into the client tool by SQL queries. Since SQL was designed as an access language to relational databases, it is not necessarily optimal for multidimensional queries. For instance, SQL can perform more complex calculations across rows than across columns. This is not a deficiency of SQL as much as a reflection of the fact that the relational model was invented to overcome a number of problems associated with database management. One of the primary problems was maintaining database integrity and ensuring consistent data updates. By storing the data in relational tables, a single piece of data is stored in one and only one place. This ensures that the database is consistently maintained and that transaction updates can be performed in a fast and efficient manner. It is often argued b y vendors of tools that provide multidimensional views of data stored in relational tables that the data is stored in an open environment and is therefore accessible.



There are many other good reasons (performance, summarization and organization of data into distinct time periods, to name three), why the data should be duplicated anyway. Thus, the implication that using a relationally based DBMS eliminates the need to duplicate the data is not valid. The vast majority of ROLAP applications are for simple analysis of large volumes of information. Retail sales analysis is the most common one. The complexity of setup and maintenance has resulted in relatively few applications of ROLAP to financial data warehousing applications such as financial reporting or budgeting.

Physical Multidimensional Databases:

The next two major OLAP architectures, MOLAP and RAP, provide their own physical multidimensional databases. These architectures assume that multidimensional models, because of their unique characteristics of sparsity and the need for potentially complex derived results, need an architecture all their own. Some of these architectures actually predate the relational model. Although all software vendors are proud of their architectures and believe they are uniquely efficient (several vendors have actually patented their algorithms), in practice, there are some common techniques which are used by many of the vendors. These techniques include:

- Mapping out zero values by compression
- Using indexes of pointers to compressed arrays of values
- Sophisticated caching algorithms

Before discussing these two architectures, it is important to understand more about how sparsity causes database 'explosion' and its implications.

Data Explosion:

It is not immediately obvious that a fully calculated hypercube is usually dozens of times, and in some cases many thousands of times, larger than the raw input data. Some would argue that this is not a problem since disk space is cheap, at least relatively speaking. Despite disk space being relatively cheap, consider what happens when a 200 MB source file explodes to 10 GB. The database no longer fits on any laptop for mobile computing. When a 1 GB source file explodes to 50 GB it cannot be accommodated on typical desktop servers. In both cases, the time to pre-calculate the model for every incremental data change will likely be many hours. So even though disk space is cheap, the full cost of precalculation can be unexpectedly large.

There is a trade-off between the time to pre-calculate and response time. Many OLAP vendors assume that multidimensional databases pre-calculate everything or nothing. As the above chart suggests, the

optimum approach might be to partially pre-calculate and only leave the less requested or smaller calculations until run time.

Real-Time Analytical Processing (RAP)

RAP takes the approach that derived values should be calculated on demand, not pre-calculated. This avoids both the long calculation time and the data explosion that occur with the pre-calculation approach used by most OLAP vendors. In order to calculate on demand quickly enough to provide fast response, data must be stored in memory. This greatly speeds calculation and results in very fast response to the vast majority of requests. Another refinement of this would be to calculate numbers when they are requested but to retain the calculations (as long as they are still valid) so as to support future requests. This has two compelling advantages. First, only those aggregations which are needed are ever performed. In a database with a growth factor of 1,000 or more, many of the possible aggregations may never be requested. Second, in a dynamic, interactive update environment, budgeting being a common example, calculations are always up to date. There is no waiting for a required pre-calculation after each incremental data change. It is important to note that since RAP does not pre-calculate, the RAP database is typically 10 per cent to 25 per cent the size of the data source. This is because the data source typically requires something like 50-100 bytes per record and possibly more. Generally, the data source stores one number per record that will be input into the multidimensional database. Since RAP stores one number (plus indexes) in approximately 12 bytes, the size ratio between RAP and the data source is typically between 12 / 100=12% and 12 / 50=24%.

Single Hypercube vs. Multicube:

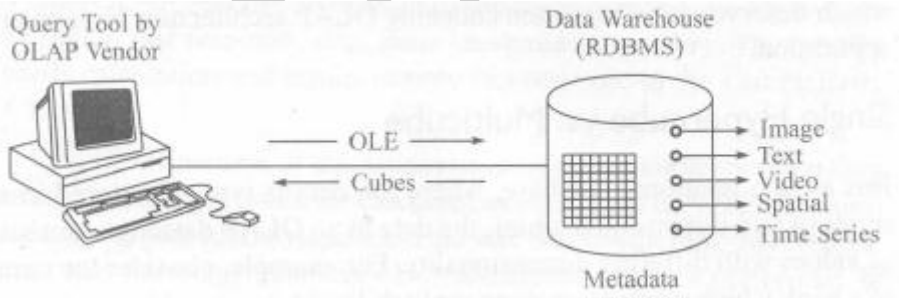
Just as in a relational database, where the data is typically stored in a series of two dimensional tables, the data in an OLAP database consists of values with differing dimensionality. For example, consider the case of a bank which wants to analyze profitability by customer and product. Certain revenues may be

dimensioned by customer, product, time, geography and scenario. But costs such as back office salaries might be dimensioned by cost type, cost center and time. Thus, a real world multidimensional database must be capable of storing data that has differing dimensionality in a single logical database that presents a single conceptual view to the user. It is not acceptable to insist that all data in the model be dimensioned by the same criteria. Putting data into an element called “no product” because the data item happens not to be dimensioned by product, merely serves to confuse the user. Products that cannot support a multicube architecture With the ability to logically relate cubes, force unreasonable compromises on the developer of sophisticated models. Not only does it make the access to the data less logical but it can have serious adverse effects on database size and retrieval efficiency.

Integrated Relational OLAP

Standard relational OLAP technology, while an impressive solution for the most complex decision support requirements, still has significant room for improvement. One problem is that the data is physically separated from the analytical processing. For many queries this is not a major issue; however, it limits the scope of the analysis that is possible. In a traditional ROLAP system, the analysis engine formulates SQL statements that it sends to the RDBMS server. It then takes the data back from the server, re-integrates it, and performs further analysis and computations before delivering the finished results to the user. This usually means that data travels over the network twice: once to the analysis server and again to the client application.

INTEGRATED ROLAP ARCHITECTURE



The next logical step in the evolution of OLAP technology is the integration of the ROLAP engine with the scalable, parallel RDBMS itself: integrated relational OLAP. With an integrated database server and cost-based optimizer for both relational and multidimensional analysis, the “universal” server can give unprecedented performance and scalability for the data warehouse.

Parallelization

One of the key problems with data warehouses is the sheer volume of data. Parallelization techniques are critical to performance; breaking down complex actions into smaller parts, each of which can be executed in parallel. The net result is faster execution. The different database vendors have achieved

significantly different results and completeness of their parallelization efforts. That is, not all parts of most database products have yet been optimized for parallel execution.

Data Partitioning

Effective data partitioning is another prong in the attack on large data volumes. Partitioning enables the database to automatically distribute portions of a table or tables into more than one logical and/or physical file, which enhances the ability for the database to parallelize operations and eases maintenance on the large data sets. Partitioning also enables the database to distribute data over multiple physical storage devices, which can be operating in full speed in parallel. The database implementation should support a variety of partitioning options, including range-based partitioning, hash-based partitioning, and round-robin partitioning, which are critical in data warehouses.

DSS indexes

An index is a means by which an RDBMS ^can very quickly access information from a table (or tables) without scanning the entire table. Indexes often speed up performance dramatically. There are several specialized indexes that enhance relational database performance for decision support applications. The most important type for OLAP-style queries is the “multi-table” index, a generalization of the star join index. A multi-table index includes columns from more than one table in the same index, effectively precomputing the joins between the tables. Since joins are one of the slower operations in query processing, this index can dramatically improve query response time. Another important new index is the “bitmap” index. Instead of using a traditional tree to represent the index, bitmaps represent the presence of a given value for a field as a “0” or “ 1” for true or false. Bitmaps efficiently index low cardinality data; that is, data that has few possibilities for the actual value.

Sampling

Sampling allows users to estimate results based on a partial representation of detailed data, and to do it in a tiny fraction of the time it would otherwise take to retrieve an answer. Results are derived using well established, time-tested statistical techniques, and query performance is orders of magnitude faster. Sampling has long been a trusted technique in academic settings to expedite trend analysis and accelerate time-critical tasks. It has also been used for decades to enable rapid election polling and to drive television ratings.

Analytical Extensibility

With the advent of object-relational techniques, it is now possible for customers and third parties to natively extend the core analytical capabilities of the database server itself. Customers can add userdefined aggregate functions and user-defined analysis functions to the database using a well-defined object programming model. Once these functions are defined, they behave just like a built-in function, with the full parallelization and performance characteristics of the core database.

Data Sparsity And Data Explosion:

The reality of data explosion in multidimensional databases is a surprising and widely misunderstood phenomenon. For those about to implement a project with considerable exposure to OLAP products, it is critically important to understand what data sparsity and data explosion are, what causes these, and how these can be avoided for, the consequences of ignoring data explosion can be very costly and, in most cases, result in project failure. Input data or base data (i.e. before calculated hierarchies or levels) in OLAP applications is typically sparse (not densely populated). Also, as the number of dimensions increase, data will typically become sparser (less dense). Data explosion is the phenomenon that occurs in multidimensional models where the derived or calculated values significantly exceed the base values.

There are three main factors that contribute to data explosion.

1. Sparsely populated base data increases the likelihood of data explosion
2. Many dimensions in a model increase the likelihood of data explosion, and
3. A high number of calculated levels in each dimension increase the likelihood of data explosion