

Architecting Essentials

(Introduction to
Architecting and
Advanced Architecting)



CONTENTS

CHAPTER 1: ARCHITECTING THE MICROSTRATEGY SCHEMA	10
Introducing the Intelligent Enterprise	10
Introduction to MicroStrategy Schema	11
Schema objects overview	11
MicroStrategy project life cycle.....	12
Gathering business requirements	12
Identifying measurements and dimensions	13
Creating a logical data model	14
Design considerations for a logical data model.....	14
Constructing the logical data model.....	15
Exercise 1.1: Designing the logical data model	18
Data warehouse physical schema design	20
Understanding the physical schema in MicroStrategy Architect	20
Physical schema components.....	21
Selecting an appropriate schema type	27
Schema design considerations	33
Exercise 1.2: Designing the data warehouse schema.....	35
Storing schema information in a metadata repository	36
Creating a project source and corresponding project	36
Exercises.....	37
Exercise 1.3: Access MicroStrategy Developer	37
Exercise 1.4: Creating the metadata database	38
Exercise 1.5: Creating the DSN for the metadata database	38
Exercise 1.6: Creating the DSN for the data warehouse	42
Exercise 1.7: Creating the metadata repository	42

Exercise 1.8: Creating the project source.....	43
Exercise 1.9: Creating the database instance	44
Exercise 1.10: Creating a project.....	45
Creating a MicroStrategy schema	47
Creating a MicroStrategy schema	47
Introduction to Architect.....	47
Accessing Architect	47
Exercise solutions	49
Exercise 1.1 solution: Designing the Logical Data Model	49
Exercise 1.2 solution: Designing the data warehouse schema.....	49
CHAPTER 2: WORKING WITH TABLES	51
Physical tables versus logical tables	51
Using layers when creating project tables.....	51
Viewing and modifying project table properties	52
Exercises	54
Exercise 2.1: Configuring the Warehouse Catalog	54
Exercise 2.2: Adding fact tables to the project.....	56
Exercise 2.3: Adding lookup tables to the project	58
CHAPTER 3: WORKING WITH FACTS.....	60
Types of facts.....	61
Types of fact expressions	62
Creating facts	63
Exercise 3.1: Creating facts	64
CHAPTER 4: WORKING WITH ATTRIBUTES AND RELATIONSHIPS	68
About attribute forms	69
Types of attributes	71
Types of attribute form expressions.....	72

Creating attributes in Architect	73
Creating attribute forms.....	84
Modifying attribute forms	87
Modifying column aliases.....	89
Viewing and modifying properties for attributes	89
Displaying attribute forms.....	92
Creating attribute relationships	93
Exercises.....	99
Exercise 4.1: Creating attributes and relationships	99
CHAPTER 5: WORKING WITH HIERARCHIES	108
All attributes in a project: System hierarchy	108
Browsing objects intuitively: User hierarchies	108
Exercises.....	118
Exercise 5.1: Creating hierarchies	118
CHAPTER 6: MANAGING THE MICROSTRATEGY SCHEMA - FACTS	125
Modifying facts.....	125
Modifying fact column aliases	126
Viewing and modifying properties for facts	126
Exercises.....	128
Exercise 6.1: Modify facts in the project	128
Exercise 6.2: Add a new fact to the project.....	131
Exercise 6.3: Modify attribute forms.....	133
Exercise 6.4: Create metrics using facts	137
CHAPTER 7: TRANSFORMATIONS.....	139
Types of transformations	139
Transformation components.....	141
Exercises.....	142

Exercise 7.1: Create a last year's transformation	142
CHAPTER 8: AGGREGATE TABLES	150
Logical table size.....	150
Data mart	151
Exercises.....	153
Exercise 8.1: Create aggregate fact table	153
CHAPTER 9: MICROSTRATEGY MULTISOURCE OPTION	165
MultiSource Option.....	165
Designating primary and secondary database instances.....	165
Support for duplicate tables.....	166
SQL generation for MultiSource reports	166
Designating primary and secondary tables	167
Selecting the optimal data source for fact tables.....	167
Selecting the optimal data source for lookup tables.....	168
Joining data from multiple data sources	168
Common use cases.....	170
Split fact tables.....	170
Split lookup tables	171
Filtering qualifications	172
Exercises.....	173
Exercise 9.1: Use the MicroStrategy MultiSource Option	173
CHAPTER 10: FACT EXTENSIONS AND DEGRADATIONS.....	182
Fact level extension.....	182
Fact degradation	182
Steps for fact degradation	184
Exercises.....	185

Exercise 10.1: Create a fact level degradation	185
Extending fact levels.....	191
CHAPTER 11: PARTITIONING	200
Partitioning fact tables in the MicroStrategy schema	200
Data import	208
Picking database tables to build MTDI Cubes.....	208
OLAP/MDX sources	209
Exercises.....	213
Exercise 11.1: Pick tables to build MTDI cubes	213
CHAPTER 12: LOGICAL VIEWS	222
Other methods for creating schema objects.....	223
Warehouse Catalog	234
CHAPTER 13: ADVANCED ARCHITECTING.....	243
Many to Many Relationships.....	243
Attribute roles	265
Create an explicit table alias definition	265
Turn on automatic attribute role recognition	265
Exercise 13.1: Configure attribute roles	266
CHAPTER 14: HIERARCHIES	277
Ragged Hierarchy	277
Split Hierarchy	286
Recursive Hierarchies	295
Exercises:.....	307
Ragged Hierarchies Overview.....	307
Split Hierarchies Overview	332
Recursive Hierarchies Overview	343
CHAPTER 15: SLOWLY CHANGING DIMENSIONS.....	357

Types of Slowly Changing Dimensions (SCDs)	357
Using a Hidden Attribute for SCDs	367
Exercises:.....	386
SCDs Overview.....	386
SCDs - Hidden Attribute Project Information	393

CHAPTER 1: ARCHITECTING THE MICROSTRATEGY SCHEMA

Introducing the Intelligent Enterprise

The Intelligent Enterprise is a data-driven organization that designs and implements Business Intelligence solutions effectively while promoting efficient use of data across your enterprise. This fosters growth and development, with a focus on data governance and alignment of strategic business goals to technology investments.

Getting there requires the right tools and structure to balance traditionally counteractive forces: agility and governance, convenience and security, ease of use and enterprise functionality, all critical capabilities that the MicroStrategy platform is positioned to support with its unique intelligence architecture. A successful Intelligent Enterprise:

- Drives adoption and success of enterprise analytics.
- Coordinates analytics implementations.
- Introducing the Intelligent Enterprise
- Maintains sound data governance and a single version of the truth, while supporting personal, feature-rich data discovery.
- Provides a formal approach to documenting processes, creating content, and performing ongoing maintenance.
- Ensures that the analytics solution is aligned with enterprise strategy.

The MicroStrategy Intelligence platform empowers organizations to analyze vast amounts of data, offer tangible solutions to business queries, build data visualizations, and enable users to share their insights anywhere and anytime.

The MicroStrategy Intelligence platform architecture includes all the required components to support an organization's analytics needs, from self-service reporting through creating enterprise applications for a complete analytical workflow.

These components include:

Metadata repositories, and the servers required to assemble metadata objects and provide core analytical processing power for any type of analytics application. The clients through which users request and receive reports, from business analysts to developers and administrators.

The tools to define and manage the analytics infrastructure.



The MicroStrategy Intelligence platform can be broken down into the following areas:

- Security
- Applications
- Data
- Schema
- MicroStrategy Intelligence platform
- Application services
- Platform services

This chapter focuses on the development of the MicroStrategy schema.

Introduction to MicroStrategy Schema

The MicroStrategy schema is a collection of logical objects that allow you to connect to your data sources and perform data analysis tasks. The schema is a representation of your users' business needs, and a communication channel to your data.

Business users leverage the MicroStrategy Analytics Platform to produce actionable intelligence from data sources such as data warehouses, mobile device management systems, enterprise directories, cloud applications, access control systems, and so on. To make this analysis possible, a MicroStrategy schema serves as the primary contact layer between your data sources and the analytics software.

To help you develop a MicroStrategy schema, you will learn to gather business information, transform that information into schema objects, diagram your existing data, and then build the required connections and logical components in MicroStrategy.

Schema objects overview

Schema objects are logical components that relate MicroStrategy application objects to data warehouse content. They are the bridge between the reporting environment and the data warehouse. As an architect, your objective is to build basic schema objects that will be leveraged by business users to create templates, filters, reports, or dossiers.

The following schema objects form the foundation of a MicroStrategy project:

- **Tables:** Correspond to physical tables stored in the data warehouse. Tables are used in a MicroStrategy project to maintain information about the tables in a data source.
- **Facts:** Relate measurable data stored in the data warehouse to the MicroStrategy reporting environment. They are usually numeric, and can be aggregated to different levels, depending on reporting needs. For example, a common fact for a retail business intelligence project is Revenue.
- **Attributes:** Relate descriptive (non-fact) data stored in the data warehouse to the MicroStrategy reporting environment. They provide business context and define the level of detail for fact analysis. For example, a common attribute for a retail business intelligence project is Store Location.
- **Hierarchies:** Group attributes to reflect their relationships to each other, or provide convenient browsing and drilling paths in the MicroStrategy reporting environment. For example, a common hierarchy is Time, which may include the Year, Month, and Day attributes.

MicroStrategy project life cycle

The MicroStrategy project life cycle is a logical progression that helps you understand the needs of business users, and positions you to create a business intelligence architecture that satisfies your organization's analytical goals. The typical phases of a MicroStrategy project life cycle are shown below:



Each phase is detailed in this chapter.

Gathering business requirements

Business requirements for a project typically include the data your organization collects, the reporting needs of your organization's various departments, and the reporting needs of your executive team.

Business requirements must be carefully collected to design an effective business intelligence architecture. Requirements are driven by the organization, its structure, its reporting needs, its preferred delivery formats, and so on. The business requirements that you document must represent the needs of business users who directly create and interact with reports.

Possible sources for business requirements include existing reports and interviews with the business staff. Rather than relying on technical staff, work with the business people who depend on reporting to understand the information they need and the terminology they employ.

For example, many business users may be interested in reporting on profits, but the criteria they use to define "profits" may vary. Some users may require information on gross profits, while others are interested in net profits. To ensure that your schema accurately reflects expectations, interview and shadow a variety of business users.

Identifying measurements and dimensions

Once business requirements have been documented, the next steps are to identify the measurements that your users want to analyze, and establish the corresponding dimensions that provide context to those measurements. Identifying these components will help you construct a logical data model and eventually create schema objects in MicroStrategy. To begin the identification

Gathering business requirements process, organize your business requirements into a Fact Qualifier Matrix, as in the following example.

Sample Fact Qualifier Matrix

Fact \ Dimension -->	Country-wide	State	City	District	Grid	Incident Group	Incident Categ	Incident Categ Dif	Time (Hierarchy)	Report Channel	Event (Hierarchy)
Incident	X	X	X	X	X	X	X	X	X	X	
Current Yr MTD Incident Count	X	X	X	X	X	X	X	X	X		X
Prior Yr MTD Incident Count	X	X	X	X	X	X	X	X	X		X
% Change from Prior Yr MTD	X	X	X	X	X	X	X	X	X		X
Current YTD Incident Count	X	X	X	X	X	X	X	X	X		X
Prior YTD Incident Count	X	X	X	X	X	X	X	X	X		X
% Change from Prior YTD	X	X	X	X	X	X	X	X	X		X
Current Yr QTD Incident Count	X	X	X	X	X	X	X	X	X		X
Prior Yr QTD Incident Count	X	X	X	X	X	X	X	X	X		X
% Change from Prior Yr QTD	X	X	X	X	X	X	X	X	X		X
Incident Count	X	X	X	X	X	X	X	X	X	X	X
Avg Incident Count for the same week number over past 5 years (A)	X	X	X			X		X			
Std dev of Incident Count for the same week number over past 5 years (B)		X	X	X		X		X			
5 year High (A+B) by week	X	X	X			X		X			
5 Year Low (A-B) by week	X	X	X			X		X			

The list of measurements is displayed in the first column, and each possible dimension appears in the subsequent columns. For each measurement, mark an X in the columns of the dimensions that your users want to analyze. This process helps you understand the relationships between the facts and attributes in your architecture.

Based on your Fact Qualifier Matrix, identify the following components:

- **Attributes:** business concepts. For example, State.
- **Facts:** measures. For example, Incident Count.
- **Hierarchies:** relationships between related attributes. For example, Country-wide, State, City, and District are attributes within a Geography hierarchy.

The next step is to group the components you identified in the Fact Qualifier Matrix into entities that will be used to create the logical data model.

Creating a logical data model

A logical data model outlines the logical relationships between the information that users want to analyze. The model helps you translate your list of attributes, facts, and hierarchies into a visual representation of the MicroStrategy schema. This diagram is a simple depiction of the flow of data in an organization, and does not reflect the physical structure of the information in the data source.

Design considerations for a logical data model

To build an effective MicroStrategy architecture, consider three fundamental concepts that impact the logical data model. These concepts include user reporting requirements, existing source data, and technical and performance considerations.

Understanding user reporting requirements

The primary measuring stick for your business intelligence architecture is its ability to satisfy your business users' reporting requirements. Consequently, the foundation of the logical data model is a solid understanding of your business users' interactions with data. An effective logical data model includes:

- All the information your users need to create and analyze reports.
- The desired granularity for each piece of information. For example, users may want to report on sales by month and by week.
- Consideration for the diverse of user needs. For example, executive users may be interested in overall business trends for the company, while a department manager may need to track sales for her assigned product area.

Once you have a firm grasp of the reporting requirements, evaluate your source data and environmental characteristics to determine whether reporting requirements can be readily supported.

Ensuring that the logical data model adequately addresses user reporting requirements is often an iterative process. Over time, changes in reporting requirements will require updates to the logical data model. Users may need to add new facts, attributes, or hierarchies or remove some components that are no longer needed or cannot be practically supported.

Evaluating existing source data

When you create a logical data model, consider whether the available source data can support user reporting requirements. Any gaps that you identify in your data will need to be rectified.

An initial review of your source systems may reveal that the data necessary to support all user reporting is available. However, if the source data is insufficient, it may be possible to use MicroStrategy to derive the required information.

For example, a company may record sales transactions by customer and city, but users may also want to analyze sales by state or region. State and region data are not part of the source data, but can be extracted based on the cities in which sales occur. Although the desired data does not exist in the source system, it can readily be inferred.

In some cases the absence of data in source systems may be a roadblock to supporting certain user reporting requirements. This can happen, for example, when the desired level of detail is not available due to limitations in data capture methodology.

Keep in mind that only information needed for reports should be part of the logical data model. Therefore, only the information needed to populate the data model should be extracted from your source systems.

Technical and performance considerations

A number of technical and performance factors can impact the logical data model design, particularly with regard to size and complexity. As the amount of information in the logical data model and the desired analytical complexity grow in scope, the demand on the analytics system also increases.

Technical issues such as database hardware and software capabilities may limit the volume or types of queries that can be supported in the reporting environment. Other factors such as network bandwidth and user concurrency may limit response time and increase system demand.

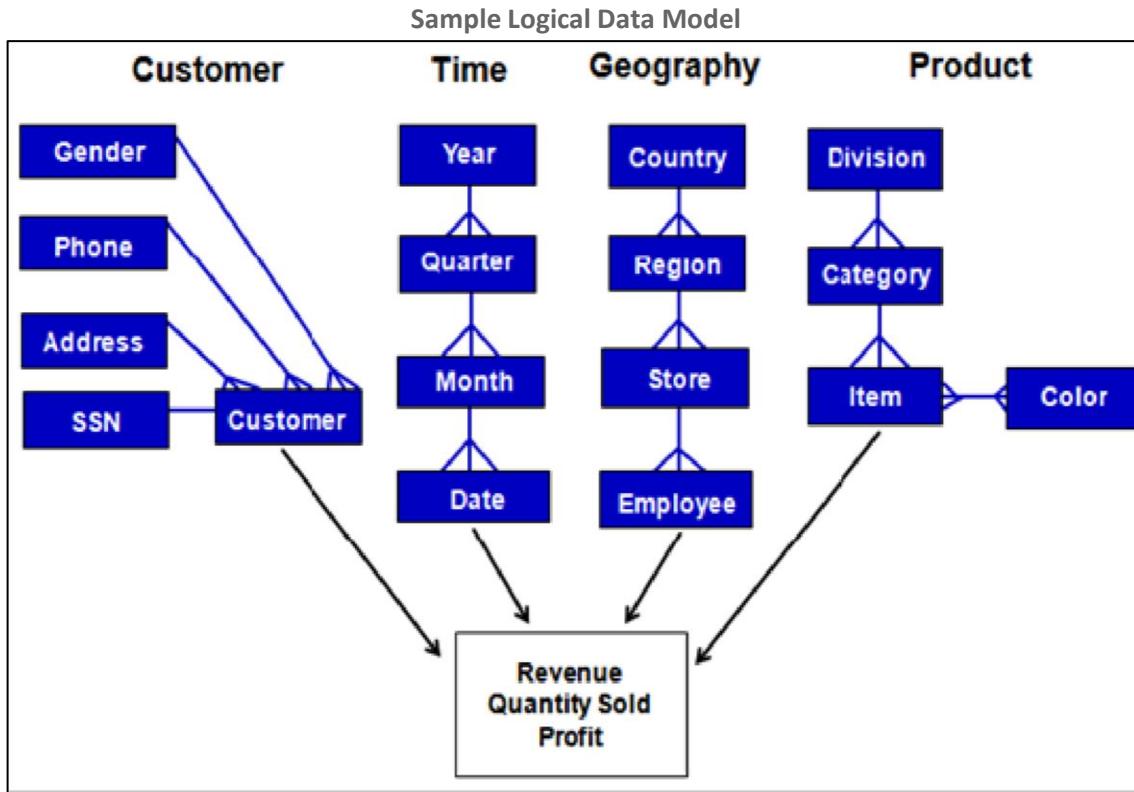
To determine the expected level of performance from your analytics solution, perform an evaluation of the available technical resources. Keep in mind that complex user reporting requirements and elaborate source data structures pose challenges to delivering a high performing solution.

For example, large projects may contain thousands of attributes, each with the potential to have hundreds, thousands, or even millions of elements. To accommodate this volume of data, you will need to consider any technical or performance limitations posed by your hardware and database software. These limitations may require you to scale back the size or complexity of the logical data model or sacrifice some user reporting requirements.

Constructing the logical data model

Based on the business requirements you gather, you will construct a logical data model that includes all the information users want to see on reports. The model helps you visualize the data that is available to you, and understand the objects you will need to create in MicroStrategy.

The model can be simple or complex, depending on the nature of your reporting requirements and the structure of available source data. The following is an example of a simple logical data model:



This model shows the relationships between the Customer, Time, Geography, and Product hierarchies. The attributes in the diagram are related to each other through a series of facts displayed at the bottom, including Revenue, Quantity Sold, and Profit.

To create a complete logical data model, include the following components:

- 1) **Facts** are measures used to analyze the organization. Fact data is typically numeric, and can generally be aggregated. Revenue, unit sales, inventory, and account balance are a few examples of common facts that may be used in business intelligence applications. Facts are displayed in the white box in the example diagram above.
- 2) **Attributes** are descriptive data that provide business context for facts. They enable users to answer questions about facts and report on various aspects of the business. Without this context, facts are meaningless. A few examples of common attributes are Customer, Region, Product Category, and Month. Attributes are displayed in the blue boxes in the diagram above.

To denote the relationship between attributes in the diagram, use Crow's Foot Notation. For example, in the diagram above, the relationship between Year and Quarter is denoted as one-to-many. This means that each year can contain one or more quarters.

Attribute Forms enable users to display descriptive information about an attribute on a report or dossier. In the diagram above, Gender, Phone, Address, and SSN are examples of attribute forms for the Customer attribute.

All attributes have an ID form. Most have at least one primary description form. Some attributes have several additional description forms that provide flexibility to describe an attribute in various ways on different reports.

Users can leverage attribute forms to display the ID for an attribute along with any number of description fields. The attribute forms map directly to columns in the data warehouse. Attribute forms must have a one-to-one relationship with other forms of the same attribute.

For example, the Customer attribute may have the following columns in a table of attribute forms:

- ID
- Name
- Home phone
- Home address
- Email
- Birth date
- Gender
- Income

A report designer can use the Customer attribute on several reports, but he may choose to display only customer names on a customer inventory report, customer names with gender and income on a marketing analysis report, and customer names and emails on a sales campaign report. This is easily accomplished through the use of attribute forms.

- 3) **Hierarchies** are groups of directly related attributes ordered to reflect their relationships. In a logical data model, hierarchies are sometimes referred to as dimensions. When you group related attributes into a hierarchy, users can easily drill to view alternate levels of detail in their reports.

Attributes are generally organized into hierarchies based on logical business areas. For example, the Time hierarchy may consist of the Day, Week, Month, and Year attributes. If the Year attribute is used on a report, an analyst can easily drill down to display measurements on the Month, Week, and Day levels of the hierarchy. Hierarchy names are displayed at the top of the diagram above.

Exercise 1.1: Designing the logical data model

Business scenario

Your goal is to design a logical data model based on the information provided in this exercise. The logical data model that you create will help you visualize the flow of data and identify the objects you will create in the MicroStrategy schema.

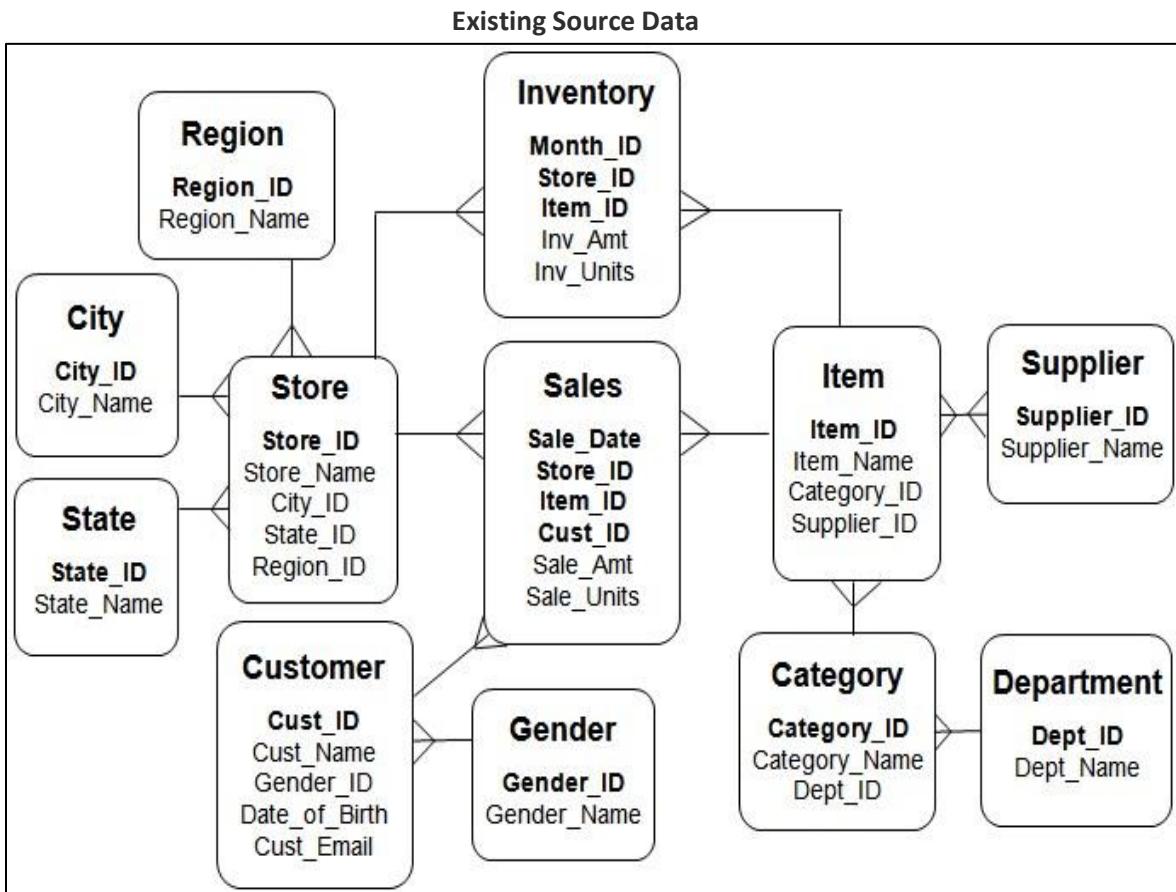
Once you complete your model, the class will review the scenario together and discuss possible solutions.

Creating a logical data model is an exercise that is open to interpretation and variability. As long as the decisions you make in constructing the logical data model consider user requirements and the nature of the underlying source data, you can develop several “right” answers. The structure of some logical data models may be preferred over others based on implementation requirements, but it is often possible to create several serviceable data models for the same scenario.

Overview

Your company captures information about various aspects of its business and stores it in a dedicated source system. The company wants to use this existing data to create a well-organized data analytics solution that enables users to understand business data.

You are new to the company and have been assigned the task of creating the logical data model that will be used to design the data analytics solution. In creating the logical data model, you need to consider the available source data displayed in the following diagram:



This diagram shows a very simple source system structure. In most business environments, source systems are much more complex, and you often have to integrate information from multiple source systems.

Create the logical data model based on the following requirements:

- Business users regularly analyze all of the available source data. Therefore, the logical data model must include everything from the source data diagram above.
- Users want to analyze data at different levels of time.
- Users want to analyze purchases by customer age.

Follow these five steps to complete this exercise:

- 1) Based on the user reporting requirements, list all the information from the source data you need to include in the logical data model.
- 2) Identify the required facts.
- 3) Identify the required attributes.

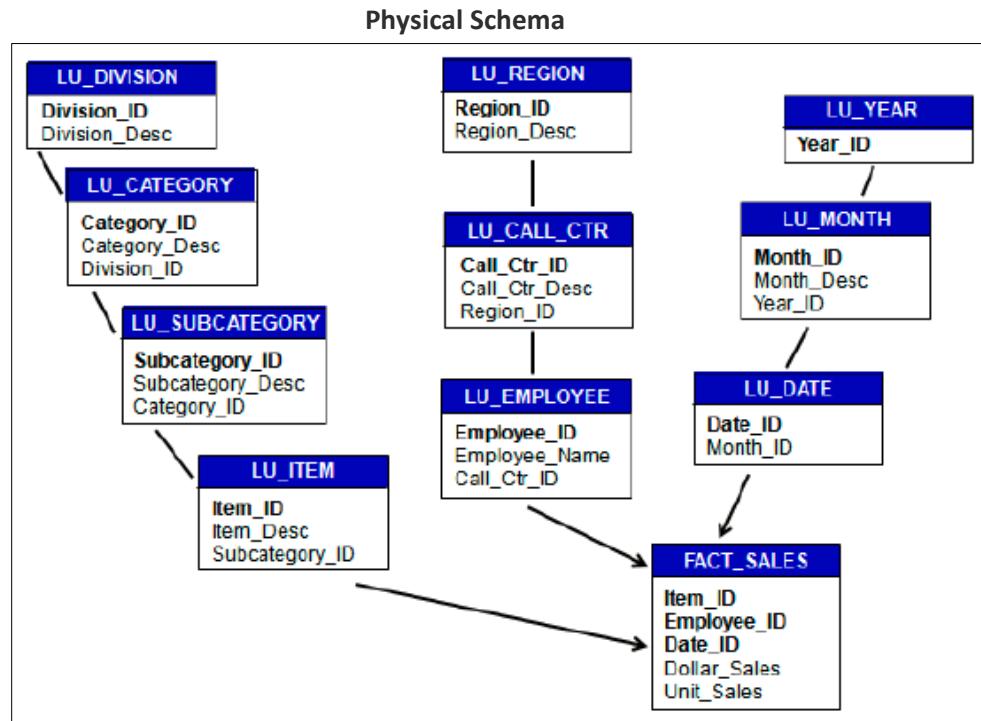
- 4) Determine the direct relationships between attributes.
- 5) Organize directly related attributes into hierarchies.

Data warehouse physical schema design

A physical schema is a detailed, graphical representation of the physical structure of a database. The physical schema helps you visualize how business data is stored in a data warehouse. You will use this information to develop your MicroStrategy schema and connect logical schema objects to your physical data source.

The physical schema design is created based on the logical data model. While the logical data model focuses on facts and attributes, the physical schema shows how the underlying data for these objects are stored in the data warehouse.

The following diagram shows a simple physical schema:



Tables in the data warehouse are displayed as rectangles, with table names in the blue headers. Within each table, the column names are displayed, with table keys in bold. The data warehouse in this example contains product, geography, time, and sales data.

Understanding the physical schema in MicroStrategy Architect

The schema objects created in MicroStrategy Architect serve as the link between the logical structure of the data model and the physical structure of the data warehouse content. When facts and attributes are created in Architect, they are mapped to specific columns and tables in the data warehouse. Therefore, understanding

the physical schema of the data warehouse is essential to creating the correct mappings between schema objects and the actual data.

Physical schema components

A physical schema includes two primary components, columns and tables. The diagram consists of a set of tables, with each table containing columns that store the actual data.

Column types

In a data warehouse, the columns in tables store fact or attribute data. The following are the three types of columns:

- **ID columns** store attribute identifiers. For example, an Employee table may have an Employee ID column that displays a unique identification number for each employee.

IDs are often numeric and generally serve as the unique key that identifies each element of an attribute. All attributes have an ID column.

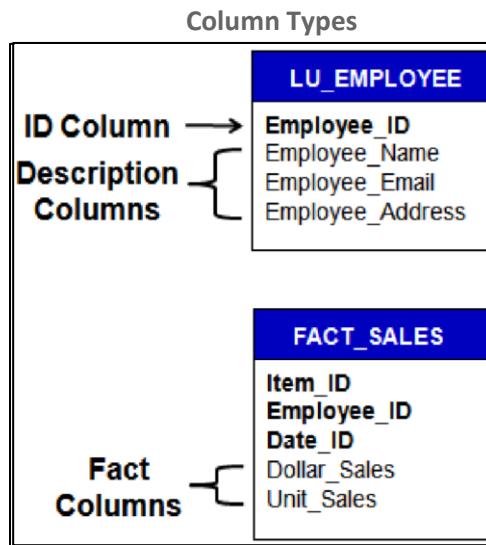
- **Description columns** store the text descriptions for attributes. For example, an Employee table may have an Employee_Name column that contains the first and last name of each employee.

Description columns are optional, but most attributes contain at least one. If an attribute has additional description columns, you can create attribute forms for each column in MicroStrategy.

When an attribute does not have a separate description column, the ID column serves as both the ID and description.

- **Fact columns** store the measures of business performance. Facts are usually numeric. For example, a Sales table may have a Dollar_Sales column that contains the total sales amount in dollars for each item sold by an employee on a specific date.

The following image displays examples of all three column types:



In the LU_EMPLOYEE table, the Employee_ID column stores the unique IDs for each employee, while the other three columns store description information about each employee, including their names, emails, and physical addresses.

In the FACT_SALES table, the Dollar_Sales and Unit_Sales columns store measured values. The remaining columns are attribute ID columns.

Table keys

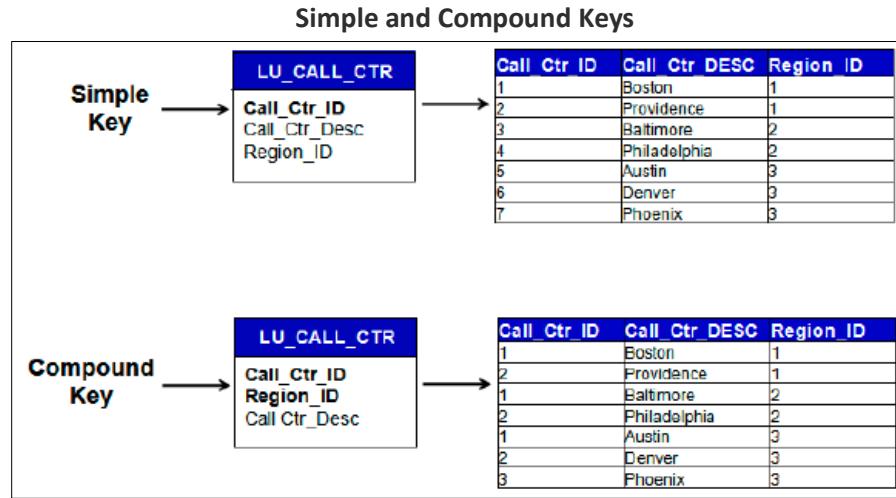
Before learning about the different types of tables in a physical schema, it is important to understand how the records in a table are identified with unique identifiers.

Every table has a primary key that consists of a unique value that identifies each distinct record (or row) in the table. There are two types of primary keys:

- **Simple keys** require only one column to uniquely identify each record within a table.
- **Compound keys** requires two or more columns to uniquely identify each record within a table.

The type of key used for a table depends on the nature of the data itself. Specific business requirements may also influence a table's key type.

The following image shows examples of simple and compound keys:



In the first example, each call center in the LU_CALL_CTR table can be uniquely identified using only the Call_Ctr_ID column. Because each call center has a unique ID, the table has a simple key.

In the second example, each call center in the LU_CALL_CTR table is uniquely identified using a combination of the Call_Ctr_ID and Region_ID columns. The table requires a compound key because call centers in different regions can have the same ID. For example, Boston and Baltimore have the same Call Center IDs, but they are in different regions. You cannot distinguish between these two call centers unless you also know their region IDs.

Simple keys are easier to work with in a data warehouse than compound keys because they require less storage space and simpler SQL queries.

Compound keys tend to increase SQL complexity and query time, and require more storage space. However, compound keys can make the ETL process less complex and more efficient. Take these factors into account when you establish key structures in the data warehouse schema.

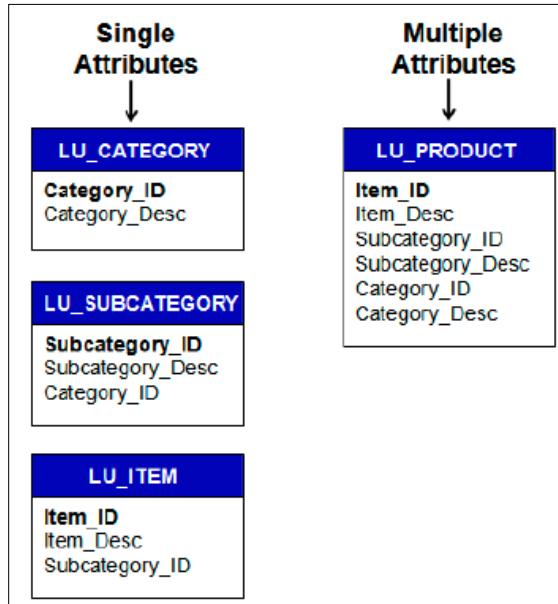
Lookup tables

Lookup tables store attribute ID and description information. They enable users to easily browse attribute data, help support data consistency, and establish a list of acceptable values. For example, a LU_CATEGORY table can be used to identify a valid list of categories.

Depending on the physical schema design, a lookup table stores information for either a single attribute or multiple related attributes.

The following image shows two examples of lookup tables:

Lookup Tables



The LU_CATEGORY, LU_SUBCATEGORY, and LU_ITEM lookup tables each contain information for only a single attribute—one of Category, Subcategory, or Item. The LU_PRODUCT table contains information for all three of these attributes.

Relationship tables

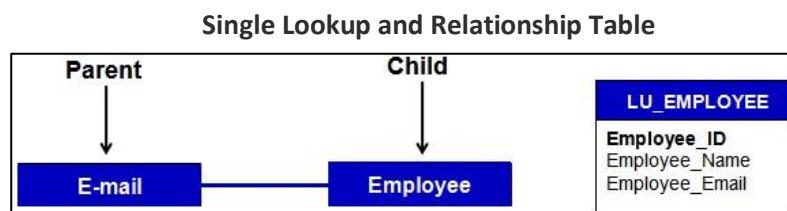
Relationship tables store information about the relationship between two or more attributes. They enable users to join data for related attributes. To map the relationship between two or more attributes, their respective ID columns must exist together in a relationship table.

Lookup tables can serve a dual function as both lookup and relationship tables in some circumstances.

Working with one-to-one relationships

Attributes that have a one-to-one relationship do not require a separate relationship table. Typically, a separate lookup table is not required for the parent attribute. Instead, the parent is placed directly in the lookup table of the child attribute to map the relationship between the two attributes.

The following example shows the use of a single lookup and relationship table for two attributes:



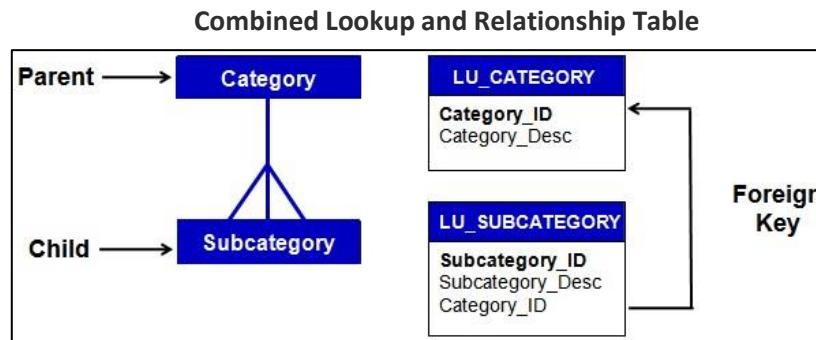
Email and Employee have a one-to-one relationship. Therefore, the LU_EMPLOYEE table can serve as both the lookup and relationship table for both attributes.

In a MicroStrategy project, often the parent attribute in a one-to-one relationship is designated as a form of the attribute it describes. To designate the parent as a separate attribute, create a separate lookup table for the parent in the data warehouse.

Working with one-to-many relationships

For attributes that have a one-to-many relationship, a separate relationship table is not necessary. The parent-child relationship can be defined by including the ID column of the parent attribute in the lookup table of the child attribute. The parent ID in the child table acts as a foreign key on the parent table, and maps the relationship between the two attributes using their respective lookup tables.

The following example shows a lookup table that also operates as a relationship table:

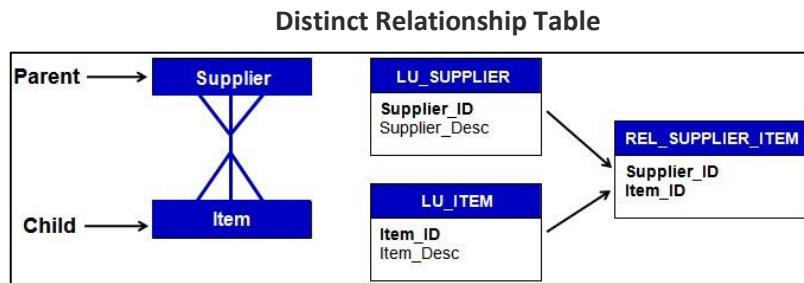


Combined lookup and relationship table

Category and Subcategory have a one-to-many relationship. Therefore, the LU_SUBCATEGORY table can serve as both the lookup table for the Subcategory attribute and the relationship table for the Category and Subcategory attributes. The Category_ID column is a foreign key that maps back to the LU_CATEGORY table.

Working with many-to-many relationships

The following example displays a many-to-many relationship and a distinct relationship table:



Supplier and Item have a many-to-many relationship. The ID column for the Supplier attribute cannot be placed in the LU_ITEM table while retaining the Item_ID column as the primary key. To map the relationship, the REL_SUPPLIER_ITEM table is created with the ID columns for both the Supplier and Item attributes. These two ID columns form a compound primary key in the relationship table.

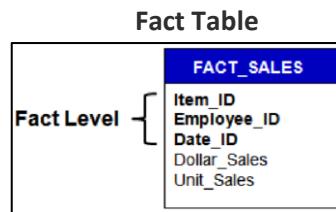
Use distinct relationship tables in a data warehouse only for attributes that have many-to-many relationships.

To analyze facts at the level of attributes that have a many-to-many relationship, the data warehouse must include a distinct relationship table, and the relationship must be included in relevant fact tables.

Fact tables

Fact tables store fact data along with attribute ID columns that describe the level of the recorded fact values. These tables enable the analysis of fact data within the context of the business dimensions or hierarchies represented by the attributes.

The following example shows a fact table:

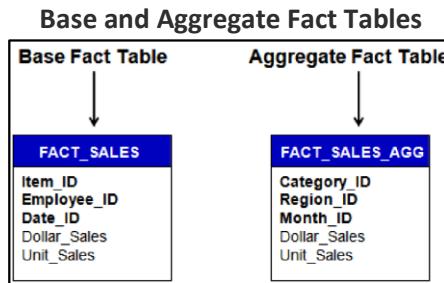


The FACT_SALES table stores dollar and unit sales measurements by item, employee, and date. These three attributes comprise the fact table level. Using this table, users can view dollar or unit sales data at any of the included attribute levels. Users can also aggregate the fact data from these levels to higher attribute levels within the same hierarchies. For example, you can aggregate the date-level dollar sales to the month level.

Base and aggregate fact tables

There are typically two types of fact tables in a data warehouse. Base fact tables store a fact or set of facts at the lowest possible level of detail. Aggregate fact tables store a fact or set of facts at a higher, or summarized, level of detail.

For example, consider the following two fact tables:



The FACT_SALES table stores dollar and unit sales data at the lowest possible level of detail—by item, employee, and date. Therefore, it is the base fact table for Dollar_Sales and Unit_Sales. The FACT_SALES_ AGG table stores dollar and unit sales data at a higher level of detail—by category, region, and month. Therefore, it is an aggregate fact table for these two facts.

Because they store data at a higher level, aggregate fact tables reduce query time. For example, a report that shows unit sales by region can obtain the result set more quickly using the FACT_SALES_ AGG table than the FACT_SALES table.

Many data warehouses contain multiple aggregate fact tables for the same fact or set of facts to increase the speed of analysis at different levels of detail.

Selecting an appropriate schema type

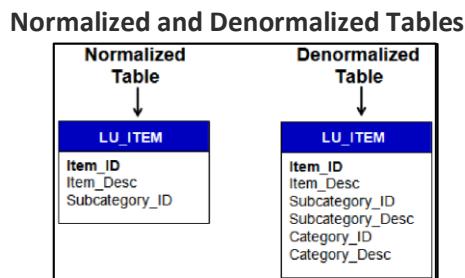
For any given logical data model, a variety of schemas can be designed to physically store the data. The type of schema design selected for the data warehouse depends on the nature of the data, how users want to query the data, and other factors unique to a project and database environment. Your physical schema design depends on the specific characteristics and needs of your organization.

Normalized versus denormalized schemas

A basic characteristic of any physical schema is its degree of normalization or denormalization.

Normalization is characterized by a schema design that does not store data redundantly. A denormalized schema design stores at least some data redundantly. Generally, tables are denormalized for performance purposes to reduce the number of joins that need to occur between tables when a query is executed.

For example, consider the following lookup tables:



The normalized LU_ITEM table contains the minimal amount of information required to store items and map their relationships to subcategories. In this scenario, even a moderately complex query will likely require the lookup table to be joined with other tables to retrieve the desired information.

The denormalized LU_ITEM table contains IDs and descriptions for item, subcategory, and categories. The data in the denormalized table is likely also stored in separate lookup tables for each attribute, which introduces redundancy in the data warehouse. Although a denormalized table is more efficient to query, it may create data integrity problems if data is not properly updated in all possible locations.

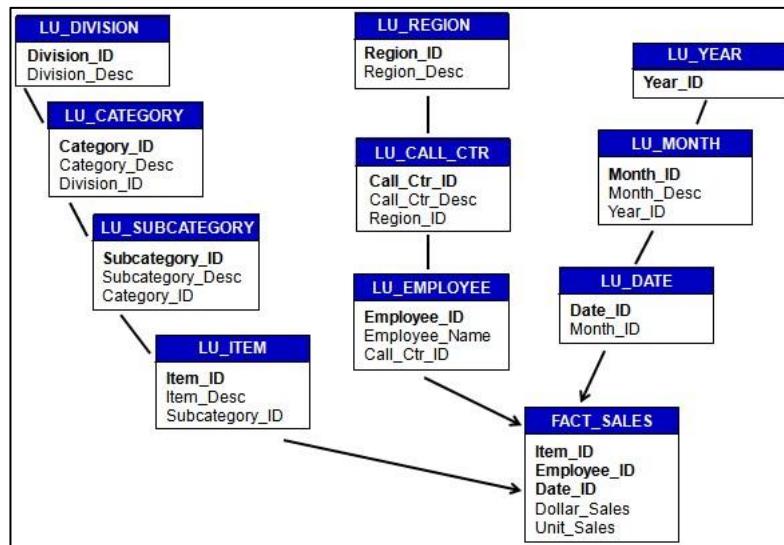
Denormalization strategies vary by the unique characteristics of each organization. Schema designs can be categorized by the degree of denormalization they employ, as outlined in the following sections.

Completely normalized schema

A completely normalized schema does not store any data redundantly. This schema design ensures data integrity through a lack of redundancy. However, query performance may be limited due to the high number of joins required to retrieve data at the desired level.

A completely normalized schema is displayed in the following example.

Completely normalized schema



For simplicity, the image above shows only one fact table as part of the schema. Schemas generally contain multiple fact tables at different levels.

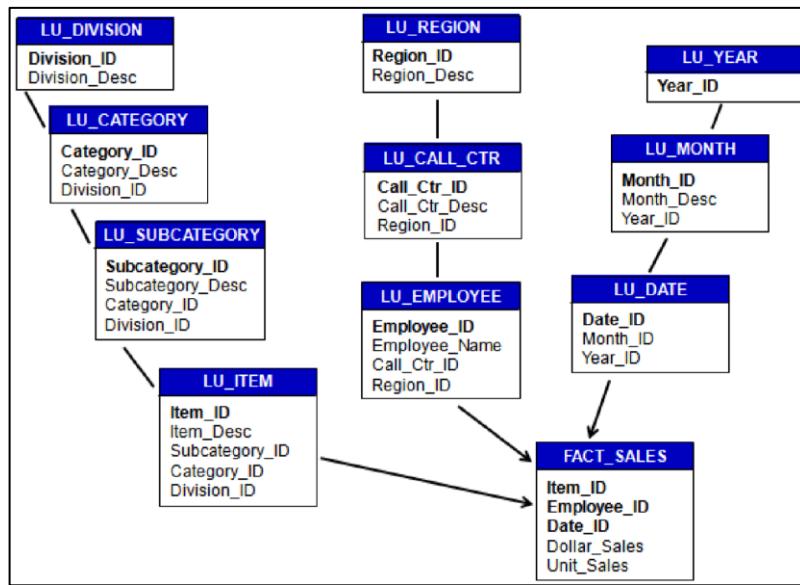
The lookup tables in this example contain only the IDs and descriptions for their respective attributes, as well as the IDs of the immediate parent attributes. For example, the `LU_EMPLOYEE` table has only three columns—`Employee_ID`, `Employee_Name`, and `Call_Ctr_ID`.

Moderately denormalized schema

A moderately denormalized schema is a compromise between the low redundancy of a normalized schema and the high analytical performance of a denormalized schema. This schema design stores the IDs of related higher-level attributes redundantly, reducing the number of joins required between tables in a normalized table. Because this schema type is less redundant than a fully denormalized schema, data integrity is better protected.

The following example shows a moderately denormalized schema:

Moderately Denormalized Schema



For simplicity, the image above shows only one fact table as part of the schema. Schemas generally contain multiple fact tables at different levels.

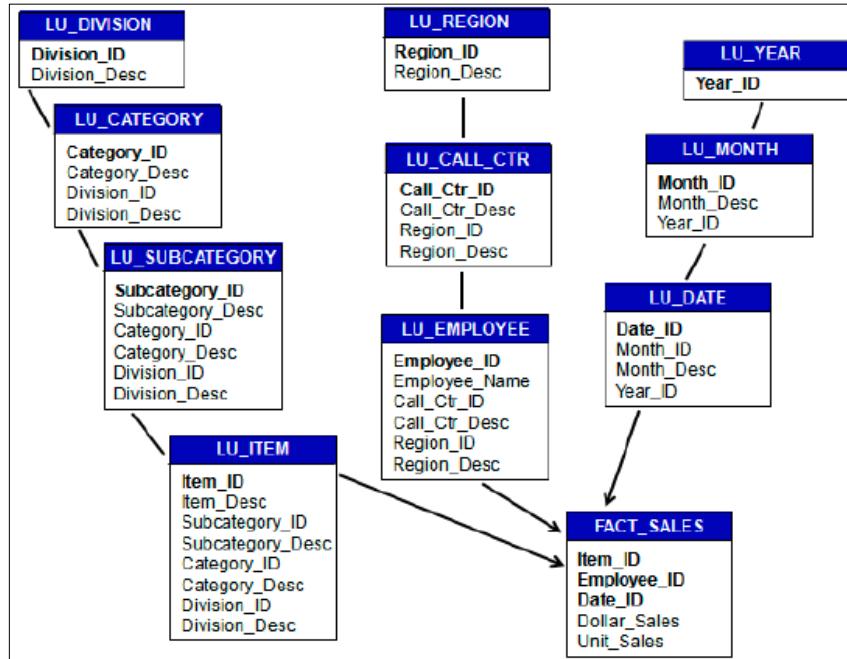
The lookup tables in this example contain the IDs and descriptions of their respective attributes as well as the IDs of all related higher-level attributes within the hierarchy. For example, the LU_EMPLOYEE table has four columns—Employee_ID, Employee_Name, Call_Ctr_ID, and Region_ID.

Completely denormalized schema

A completely denormalized schema stores the ID and descriptions of related higher-level attributes redundantly, reducing the number of joins required between tables. As a result, the performance of analytical queries is increased, but the potential for data integrity issues is increased. Additionally, the ETL process requires more time to update redundant information.

The following image shows a completely denormalized schema:

Completely Denormalized Schema

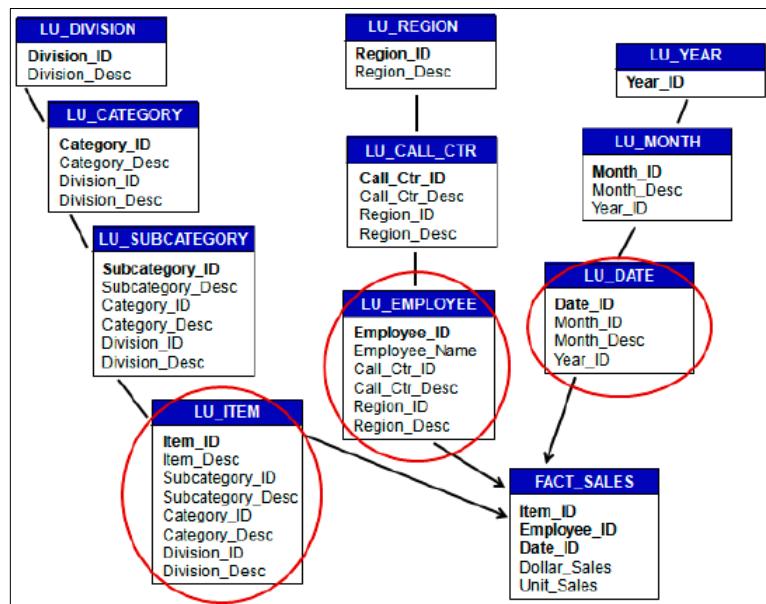


For simplicity, the image above shows only one fact table as part of the schema. Schemas generally contain multiple fact tables at different levels.

The lookup tables in this example contain the IDs and descriptions for their respective attributes, as well as the IDs and descriptions of all related higher-level attributes within the hierarchy. For example, the LU_EMPLOYEE table has six columns—Employee_ID, Employee_Name, Call_Ctr_ID, Call_Ctr_Desc, Region_ID, and Region_Desc.

In a completely denormalized schema, the lowest-level lookup tables for each hierarchy contain all the information found in the higher-level lookup tables, as in the following example.

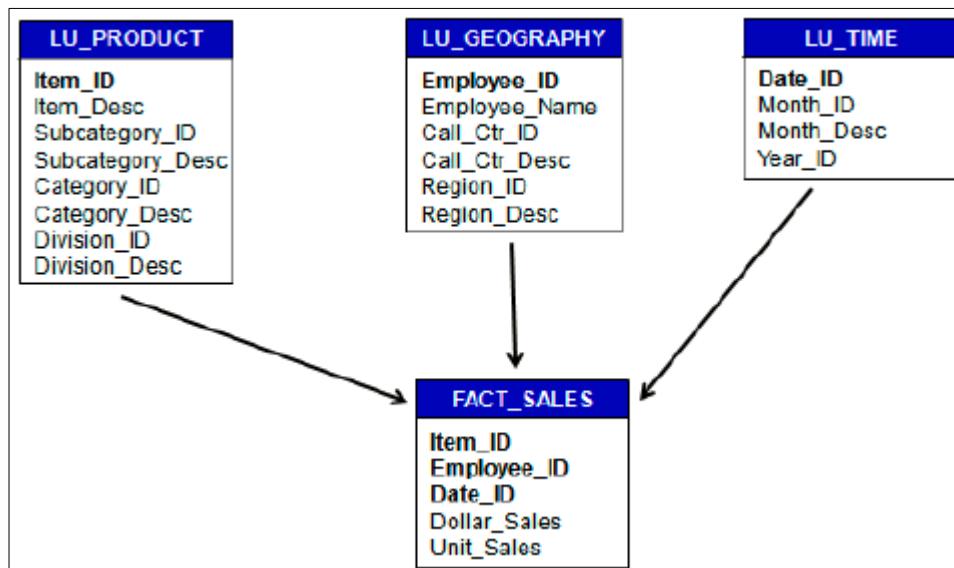
Lowest-Level Lookup Tables



The LU_ITEM, LU_EMPLOYEE, and LU_DATE tables contain all the information in their respective hierarchies. The other, higher-level lookup tables do not provide any additional information that cannot be obtained from these three tables.

To simplify a denormalized schema, you can eliminate all the higher-level lookup tables to create a schema that has only one table for each hierarchy. This is often referred to as a star schema, as displayed in the following example.

Star Schema



Although a completely denormalized schema can be simplified to a single table per hierarchy, there are advantages to retaining the higher-level lookup tables.

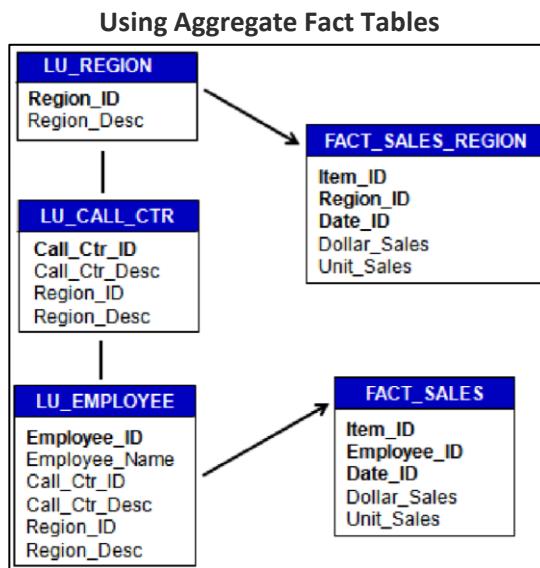
First, depending on how table keys are created in a star schema, higher-level attributes can be browsed more efficiently using the higher-level lookup tables.

For example, to create a report that shows a list of categories, the MicroStrategy SQL Engine performs a SELECT DISTINCT query on the LU_PRODUCT table because the table does not contain a distinct list of categories. Unfortunately, SELECT DISTINCT queries are more resource-intensive than SELECT queries.

If the schema contained the LU_CATEGORY table, The Engine could obtain the same result by performing a simple SELECT on the table. The performance benefits of the completely denormalized schema are completely negated by using the SELECT DISTINCT query.

Second, higher-level lookup tables are required to take advantage of aggregate fact tables in the schema.

For example, consider the aggregate fact table in the following schema:



The FACT_SALES table stores dollar and unit sales at the item, employee, and date level. The same data is stored at the item, region, and date level in the FACT_SALES_REGION aggregate fact table. In this scenario, you would retrieve Sales by Employee from the FACT_SALES table.

If you wanted to find Sales by Region, you would query either the FACT_SALES or FACT_SALES_REGION tables. Querying the FACT_SALES_REGION table is more efficient since the data is already aggregated to the region level. However, after the MicroStrategy Engine obtains the sales data from the FACT_SALES_REGION table, it has to join to a lookup table to retrieve the corresponding region descriptions.

If the join is performed on the LU_EMPLOYEE table, a distinct list of regions would not be returned. Instead, the regions are repeated for each employee in each region. As a result, the Engine would join to each region for every occurrence in the LU_EMPLOYEE table, creating duplicate sales counts. The result set would contain inflated sales values.

If the join is performed on the LU_REGION table, a distinct list of regions would be found, and the Engine would return the correct sales values. This scenario demonstrates the need to retain higher-level lookup tables when using an aggregate fact table.

Summary of schema types

The following table provides a comparison of the characteristics of each schema type:

Schema Type	Lookup Table Structure	Advantages	Disadvantages
Completely normalized schema	Contains attribute ID and description columns, as well as the ID column of the immediate parent attribute	Requires minimal storage space and no data redundancy	Requires more joins for queries on higher-level attributes
Moderately denormalized schema	Contains attribute ID and description columns, as well as the ID columns of all related higher-level attributes	Significantly reduces the number of joins necessary for queries on higher-level attributes	Requires slightly more storage space and some data redundancy
Completely denormalized schema	Contains attribute ID and description columns, as well as the ID and description columns of all related higher-level attributes	Further reduces the number of joins necessary for queries on higher-level attributes	Requires significantly more storage space and the greatest degree of data redundancy

The best structure for any data warehouse is often a combination of schema types. For example, you may choose to normalize one hierarchy, and completely denormalize another. You may even need to normalize and denormalize various tables within the same hierarchy to fit your needs.

Schema design considerations

The ultimate goal of the data warehouse schema design process is to create a data warehouse structure that satisfies the reporting needs of your business users. As you develop your data warehouse schema design, consider the following factors:

- User reporting requirements
- Query performance
- Data volume
- Database maintenance

Compromises and trade-offs are integral to the schema design process. The following topics describe how each factor influences your design.

User reporting requirements

The structure of the data warehouse schema must satisfy user reporting requirements. Understanding the information users need to see on reports helps you determine the query profile—the types of queries users execute and the frequency of execution.

For example, understanding the query profile can help you determine whether normalization is appropriate. If users infrequently query specific data in your warehouse, you may target that data for normalization. On the other hand, data that users frequently query may be good candidates for denormalization.

Query performance

Query performance is an indication of the response time for a given query. To ensure user satisfaction with your BI architecture, query performance must be considered when developing your data warehouse schema. To design the schema, you must understand how queries function. For example, granular or detailed queries may target large volumes of data, which can degrade performance. To address this problem, you may choose to denormalize tables to reduce the number of joins and improve performance.

Data volume

The volume of data stored for each attribute or hierarchy can help you design your warehouse schema and meet user performance expectations. Attributes and hierarchies with large data volumes are often candidates for denormalization, and attributes with lower data volumes may benefit from a more normalized schema design. For attributes with low data volumes, multiple joins between low-volume tables may produce better results than a small number of joins between high-volume tables.

Database maintenance

Database maintenance tasks include the processes required to update and maintain the information in your data warehouse. As you design your data warehouse schema, consider the resources required to maintain the chosen design.

Often, designs that provide greater flexibility in terms of query performance and detail also require extensive maintenance work. In general, maintenance requirements increase with the degree of normalization in your warehouse. Because denormalized schemas store data redundantly, updates must be performed in multiple tables. As you develop your schema design, balance the performance benefits of denormalization with its associated maintenance requirements.

For example, you may choose to mitigate the maintenance overhead associated with full denormalization by partially denormalizing specific tables.

Another aspect to consider with regard to database maintenance is the complexity of the Extract, Transform, and Load (ETL) process for converting data from source system structures to the data warehouse. Conversion complexity increases with the number of tables involved. As part of your schema design, consider the resources required to complete your ETL process.

Exercise 1.2: Designing the data warehouse schema

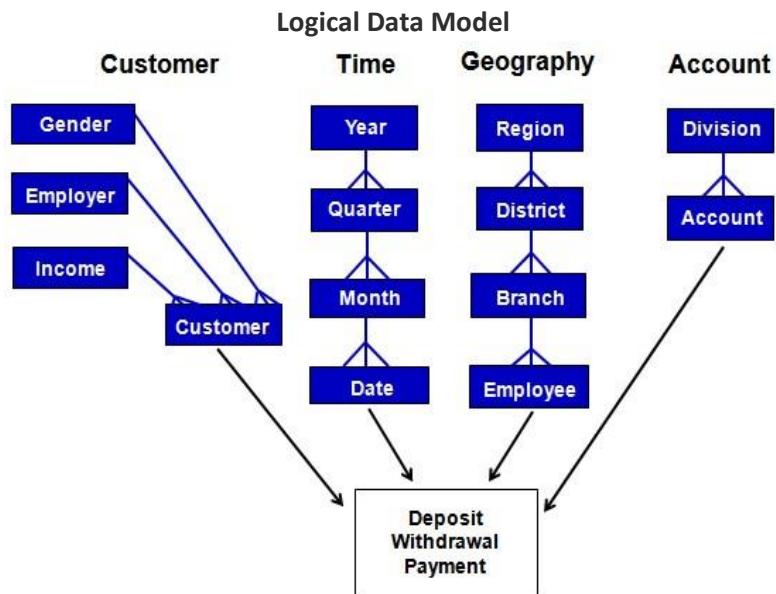
Business scenario

Your instructor will provide guidelines for completing this business scenario in groups. After you create your data warehouse schema, the class will review the scenario together and discuss possible solutions.

There is rarely a single, correct data warehouse schema. There are often several alternatives. As long as the decisions you make in structuring the physical schema are aligned with the environment described in the overview, you can develop multiple “right” answers.

Overview

You work for a bank that is building a data warehouse to help analyze its business. The following image shows the logical data model that has been created based on business requirements:



The bank wants to use this logical data model to design the data warehouse schema. You have been assigned the task of creating the schema for the Geography and Account hierarchies. As you develop the physical schema, consider the following factors to select an optimal design for each hierarchy:

- The bank evaluates and makes organizational changes to its regions and districts at least three times each year, which results in frequent updates to the relationship between those two attributes.
- The bank plans to open 50 new branches this year.
- The bank currently has 7 regions, 40 districts, and 1500 branches.

- The bank has 4 divisions with a total of over 5 million accounts.
- The source system database that the bank uses to capture transactions uses a normalized schema.
- Users execute a large number of queries to view data at the branch level. They also execute select queries to view data at the region and district levels.
- Users execute a variety of queries at both the division and account levels.
- Detailed, account-level queries often degrade performance in the current reporting system. As part of the new design, the bank wants to address the performance issues posed by these queries.

The source system contains the following minimum information for each attribute in the Geography and Account hierarchies:

- ID column that uniquely identifies each element
- DESC column that provides a text description for each element

For the Employee attribute, the source system also stores each employee's Social Security Number, home address, and home phone number.

As you design the data warehouse schema, consider the following factors:

- User reporting requirements
- Query performance
- Data volume
- Database maintenance

Storing schema information in a metadata repository

Now that you have created a logical data model and a physical data warehouse schema, you will use that information to develop the MicroStrategy schema in the metadata repository and connect to your data.

The MicroStrategy metadata contains the required information for a MicroStrategy project to function, including data warehouse connection information, project settings, and MicroStrategy object definitions.

Creating a project source and corresponding project

The metadata repository contains at least one project, which serves as a security boundary. For example, you may create one project for HR staff to report on sensitive employee data, and another project for business staff to analyze sales data. The project connects to data sources that house the information business users want to see on reports. A project also contains all objects related to an analytics solution – schema objects, reports, dossiers, cubes, filters, prompts, custom groups, metrics, and so on.

Each project is stored in a project source, which represents the connection to your metadata. A three-tier project source allows you to leverage Intelligence Server features like MicroStrategy Web, while a two-tier project source provides a limited feature set through a direct connection to the metadata.

To connect your project source to your data warehouse, you create a database instance, which houses a database connection and default database login. This allows the projects contained within your project source to communicate with your data warehouse.

In the following set of exercises, you will create a new project source called My Tutorial Project Source, and then create a new project called My Demo Project.

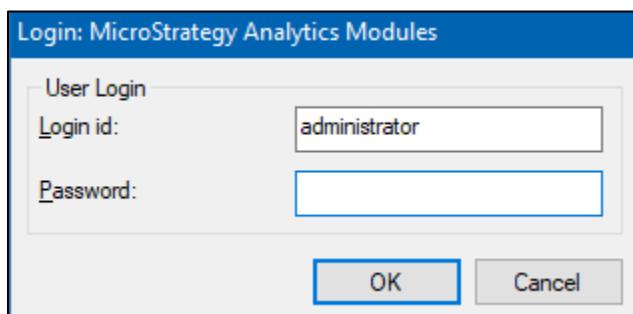
You will perform the following high-level steps:

- 1) Access MicroStrategy Developer
- 2) Create an empty metadata database
- 3) Create the DSN for the metadata database
- 4) Create the DSN for the data warehouse
- 5) Create the metadata repository structure
- 6) Create the project source
- 7) Create the database instance
- 8) Create the project

Exercises

Exercise 1.3: Access MicroStrategy Developer

1. On the **Start Menu**, type **Developer** and hit **Enter**
2. If you are prompted to login, enter **administrator** as your **Login ID** and leave the password textbox blank. Click **OK**



Exercise 1.4: Creating the metadata database

1. Create an empty database, name “**empty_shell**” in the SQL Server

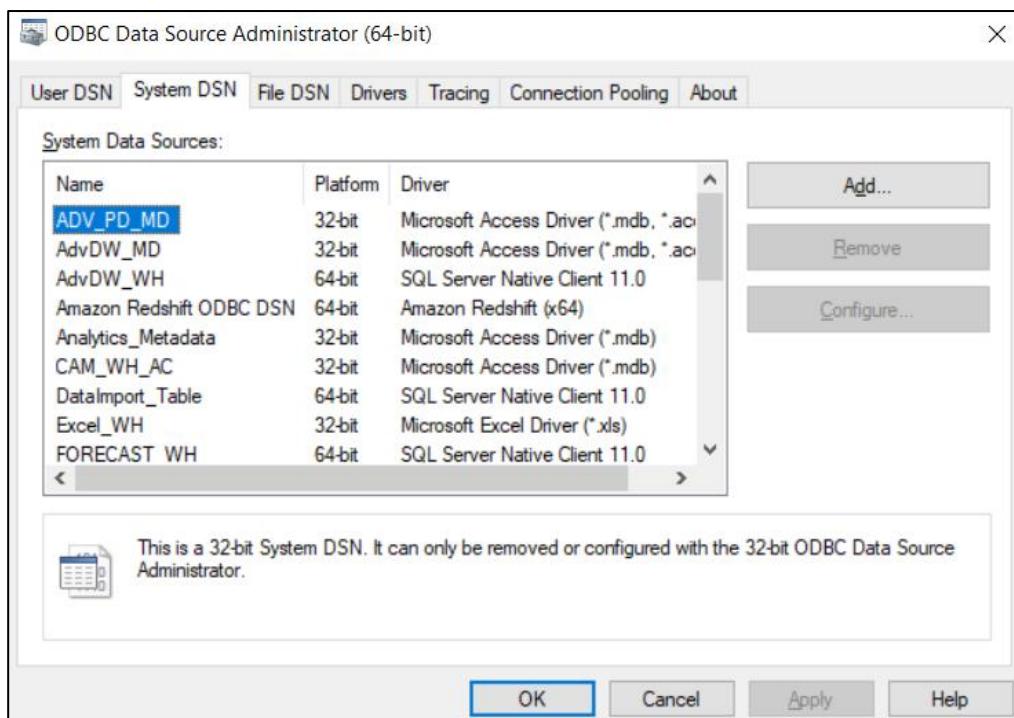
Exercise 1.5: Creating the DSN for the metadata database

Using Microsoft ODBC Administrator, you will create a DSN named **My_Tutorial_Metadata** that will be used to connect to the **empty_shell** database.

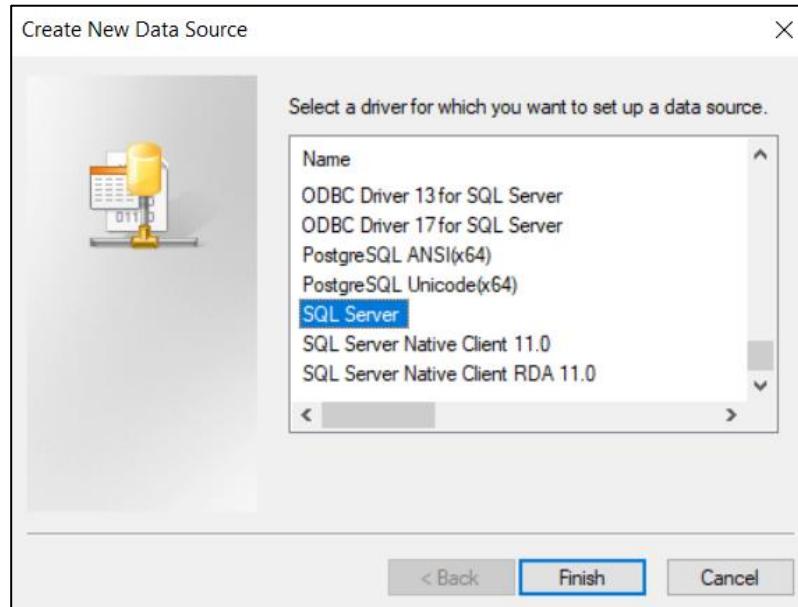
In a subsequent exercise, you will use the MicroStrategy Configuration Wizard to create the metadata repository structure in the **empty_shell** database.

Create the DSN for the metadata

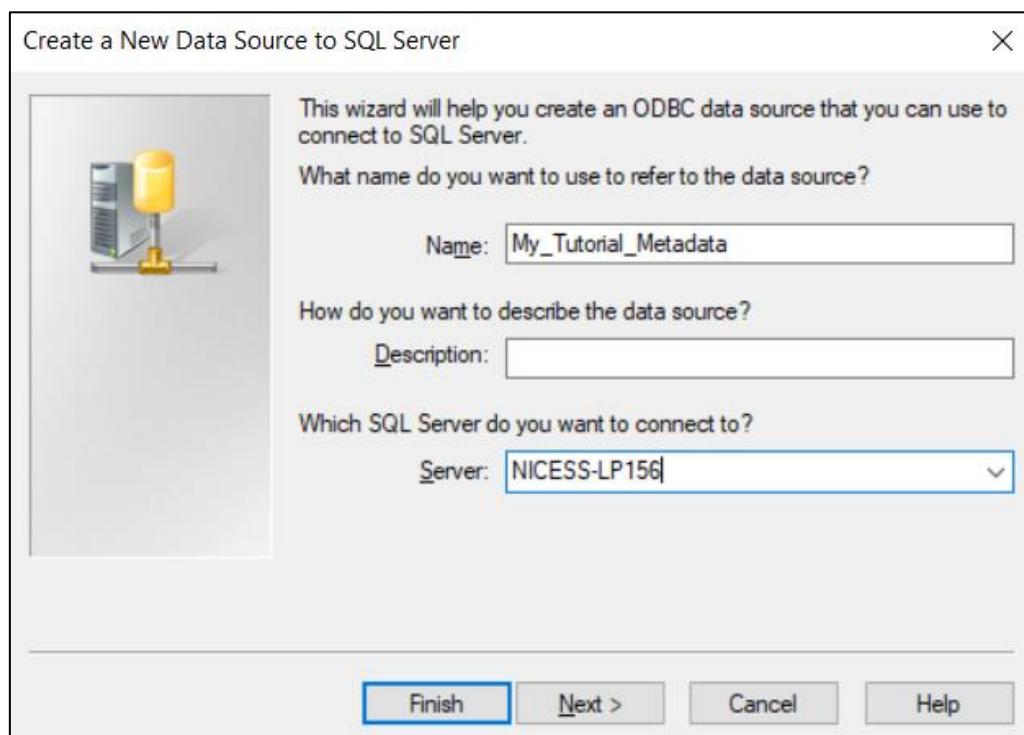
1. On the Windows machine, click the **ODBC Data Sources (64 bit)** and select **System DSN** tab.



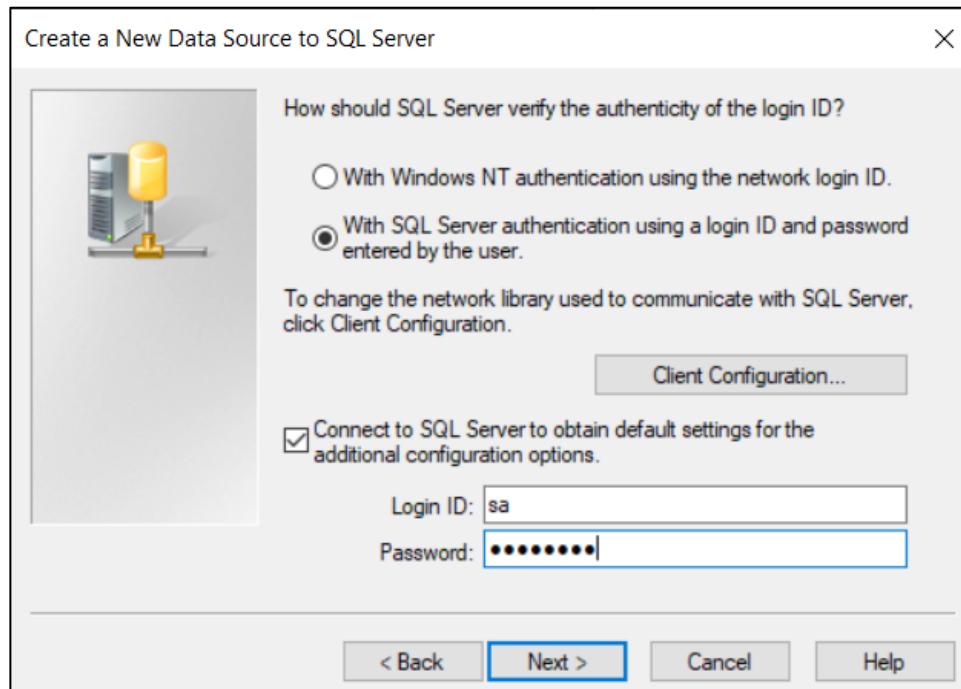
2. Click **Add**
3. Select **SQL Server** from the list and click **Finish**



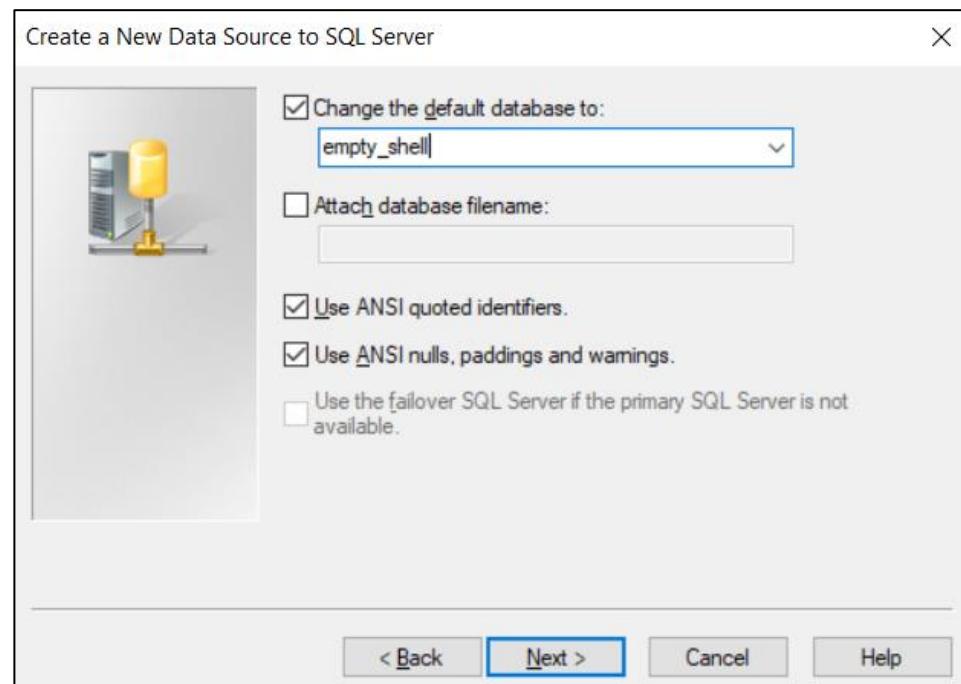
4. In the **Name** box, type **My_Tutorial_Metadata** and add the **SQL Server** details. Click **Next**.



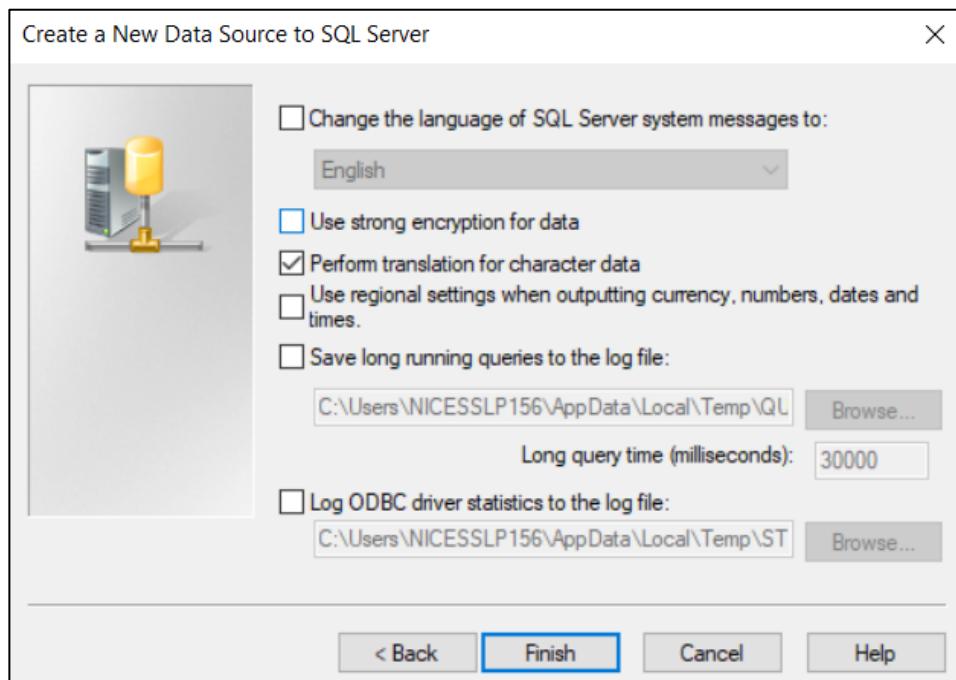
5. Enter the **SQL Server Authentication** details and click **Next**



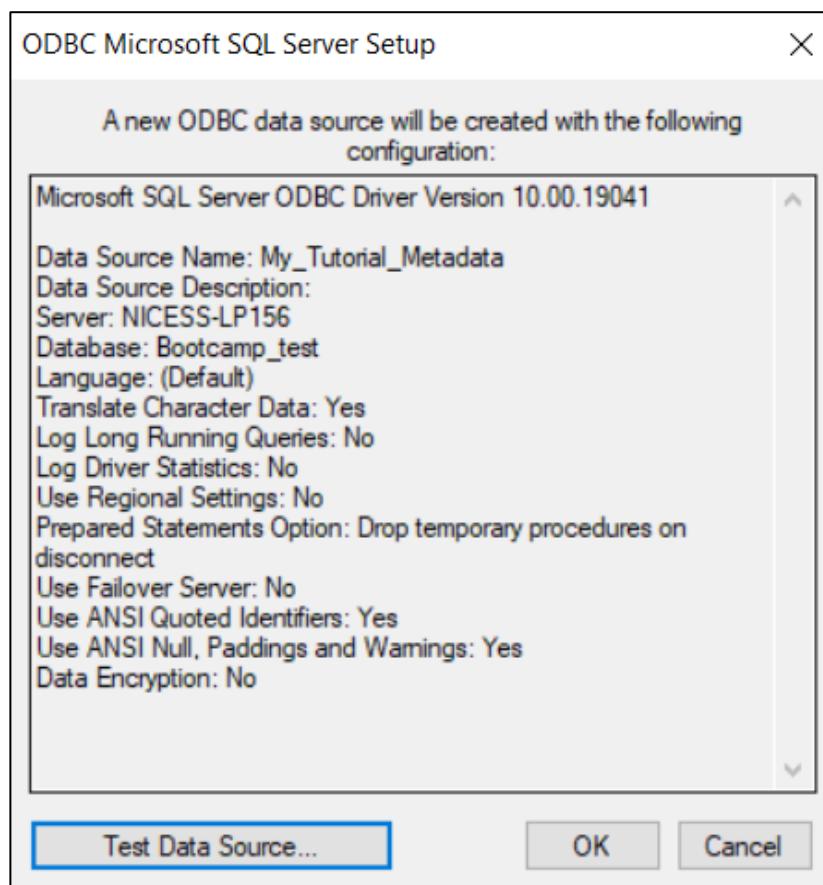
6. Check the **Change the default database to** check box and select the **empty_shell** database.
Click **Next**



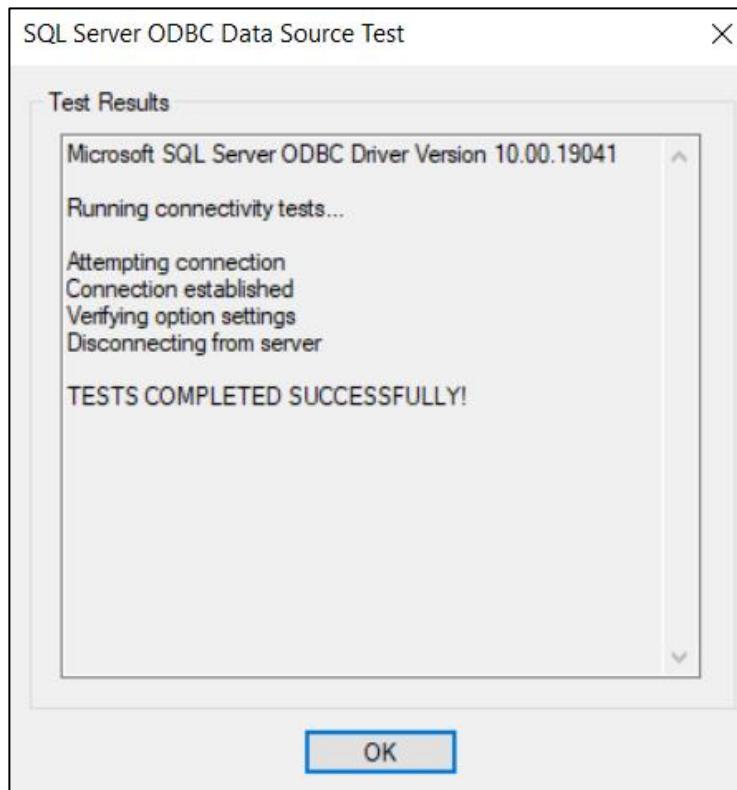
7. Click **Finish**



8. Click the **Test Data Source** to test the connection



9. Click **OK** once the connection gets established



Exercise 1.6: Creating the DSN for the data warehouse

In addition to the DSN for your metadata database, you must also create a DSN to connect to your data warehouse.

In this exercise, you will create a DSN named TUTORIAL_WH_DSN to connect to the tutorial_wh database in MSSQL.

Create the DSN for the data warehouse

Please follow the above steps

Exercise 1.7: Creating the metadata repository

Now that you have established a DNS connection to an empty metadata database, you will create the structure of the metadata repository.

In this exercise, you will use the MicroStrategy Configuration Wizard to create the metadata repository.

To create the metadata repository

1. On the Windows machine, click the Windows menu on the task bar.
2. Click **MicroStrategy Tools**, and then click **Configuration Wizard**. The MicroStrategy Configuration Wizard opens.
3. On the Welcome Page of the Configuration Wizard, select **Create Metadata, History List and Enterprise Manager Repositories**. Then click **Next**.
4. In the Repository Configuration: Repository Types window:
 - a) Clear the **History List Tables** check box.
 - b) Clear the **Statistics & Enterprise Manager Repository** check box.
5. Click **Next**.
6. In the Repository Configuration: Metadata tables window:
 - a) In the **DSN** drop-down list, select **My_Tutorial_Metadata**.
 - b) In the **User Name** box, type **sa**.
 - c) In the **Password** box, type **training**
7. Click **Next**.
8. In the Summary window, click **Apply**.
9. Upon the completion of metadata repository configuration, in the Configuration Wizard, click **Finish**, then click **Exit**.

Exercise 1.8: Creating the project source

A project source is a pointer to a metadata database. It either connects directly to a data source name (DSN) that points to the appropriate database location (a two-tier project source, or direct connection), or it connects to an instance of MicroStrategy Intelligence Server, which points to the metadata database (a three-tier project source, or server connection).

In this exercise, you will connect to the metadata database in MicroStrategy Developer by creating a project source. You will use the Project Source Manager in MicroStrategy Developer to create a new, default two-tier project source named My Tutorial Project Source that connects directly to the My_Tutorial_Metadata database.

To create the project source

1. On the Windows desktop, click the **Developer**.
2. In the Login window, click **Cancel**.
3. From the **Tools** menu, select **Project Source Manager**.
4. Click **Add**.
5. In the Project Source Manager, complete the following:
 - a. In the **Project source** box, type **My Tutorial Project Source**.
 - b. In **Connection mode**, select **Direct**.
 - c. In **ODBC DSN**, select **My_Tutorial_Metadata**.
 - d. In the **Login id** box, type **sa**.
 - e. In the **Password** box, type **training**.
6. Click **OK** twice. My Tutorial Project source appears in the Folder List.

If you want to test access, expand **My Tutorial Project Source** and log in using the default MicroStrategy login of **administrator** with blank (no) password. Since the metadata is currently empty, no projects are returned at this point. You will create a new project in a subsequent exercise.

Exercise 1.9: Creating the database instance

To connect your MicroStrategy project source to your data warehouse, create a database instance, database connection, and default database login.

In this exercise, you will use the Database Instances Manager in MicroStrategy Developer to create a database instance named TUTORIAL WH DB Instance. As part of the database instance, you will also create a database connection named TUTORIAL WH DB Connection and a database login named TUTORIAL WH DB Login.

Create the database instance

1. To open the Database Instances Manager, in Developer, log in to **My Tutorial Project Source** using the default credentials for a new project source, with the login ID of **administrator** and blank (no) password.

2. Because you have not created any projects yet, a message states that no projects were returned for the project source. Click **OK**.
 3. In the My Tutorial Project Source, expand **Administration**.
 4. Expand **Configuration Managers**.
 5. Right-click **Database Instance**, point to **New**, and select **Database Instance**.
 6. To create the database instance, in the Database Instances window, on the General tab, in the **Database instance name** box, type **TUTORIAL WH DB Instance**.
 7. From the **Database connection type** drop-down list, select **Microsoft SQL Server**.
 8. Create the database connection
 9. To create the database connection, under **Database connection (default)**, click **New**.
 10. In the Database Connections window, on the General tab, in the **Database connection name** box, type **TUTORIAL WH DB Connection**.
 11. In the **Local system ODBC data sources** list, click **TUTORIAL_WH_DSN**.
 12. Enter the database login information
 13. Under **Default database login name**, click **New**.
 14. In the Database Logins window, in the **Database login** box, type **TUTORIAL WH DB Login**.
 15. In the **Login ID** box, type **sa**.
 16. In the **Password** box, type **training**.
 17. Click **OK**.
 18. Complete the database instance creation
 19. In the Change Comments window, in the **Comments** field, type **Added a login to the Tutorial WH for user sa**. Click **OK**.
 20. In the Database Connections window, click **OK**.
 21. In the Change Comments window, in the **Comments** field, type **Added a connection to the Tutorial WH**. Click **OK**.
 22. In the MicroStrategy Developer - Server Administration window, click **OK**.
 23. In the Database Instances window, click **OK**. The Tutorial WH DB Instance is displayed in Database Instances.
- Keep Developer open for the next exercise.

Exercise 1.10: Creating a project

A MicroStrategy project serves as a security boundary that includes appropriate information for a specific group of business users.

You will now create a new project named My Demo Project in My Tutorial Project Source.

To create a project

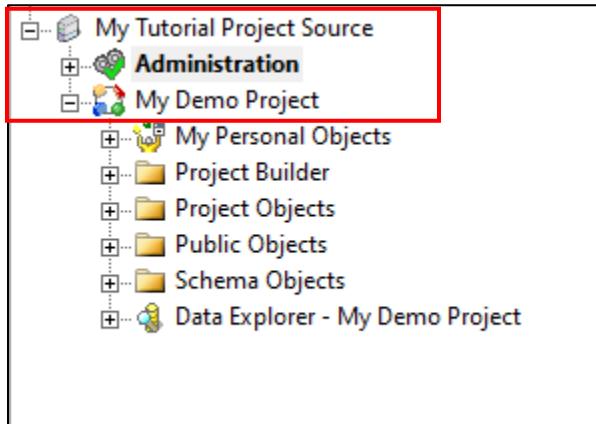
1. To open the Project Creation Assistant, in Developer, in My Tutorial Project Source, from the **Schema** menu, select **Create New Project**.
2. In the Project Creation Assistant, click **Create project**.
3. In the New Project window, type **My Demo Project** as the project name.

4. Accept all other default values and click **OK**.

MicroStrategy Architect populates the metadata tables with initial project data. This process takes a few minutes. When project initialization is complete, you return to the Project Creation Assistant. The Create Project step should have a green check mark next to it.

5. In the Project Creation Assistant, click **OK**.

6. In the message window, click **OK**. My Demo Project is added to My Tutorial Project Source.



Creating a MicroStrategy schema

Now that you have created your project, you can start developing your MicroStrategy schema to create logical objects that relate to the information in your data warehouse.

Creating a MicroStrategy schema

After you create a project in MicroStrategy and link a database instance, follow the Schema Creation Workflow to create schema objects. You will perform these schema creation tasks in the Architect tool.



Introduction to Architect

Architect is a visual, drag-and-drop tool that streamlines project design tasks. Through Architect, you can create, modify, remove, or configure various settings for the following schema objects:

- Tables
- Facts
- Attributes
- User hierarchies

Although you can perform most project design tasks in Architect, some processes may require additional tools. These tasks include creating a project, mapping

Introduction to Architect attributes or facts to partitioned tables, and creating fact extensions. These topics and their corresponding tools are addressed later in the course.

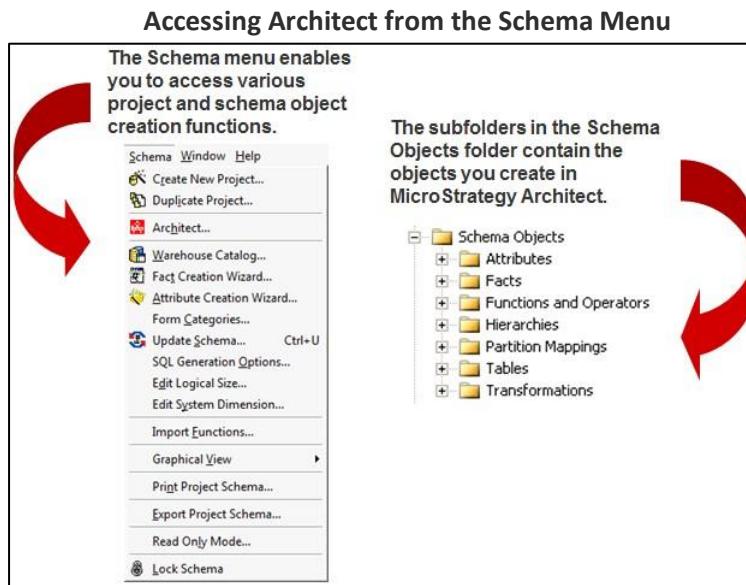
For bulk project creation, an alternative to Architect is the Project Creation Assistant, which is a collection of wizards that lead you through the project creation process.

Accessing Architect

Architect is a visual tool that streamlines the project creation process.

To open Architect

1. In Developer, log in to the project source that contains the project you want to modify.
2. Open the project to be modified.
3. From the **Schema** menu, select Architect.
4. In the Read Only window, select whether you want to enter Architect in **Read Only** or **Edit** mode.
5. Click **OK**. MicroStrategy Architect opens.



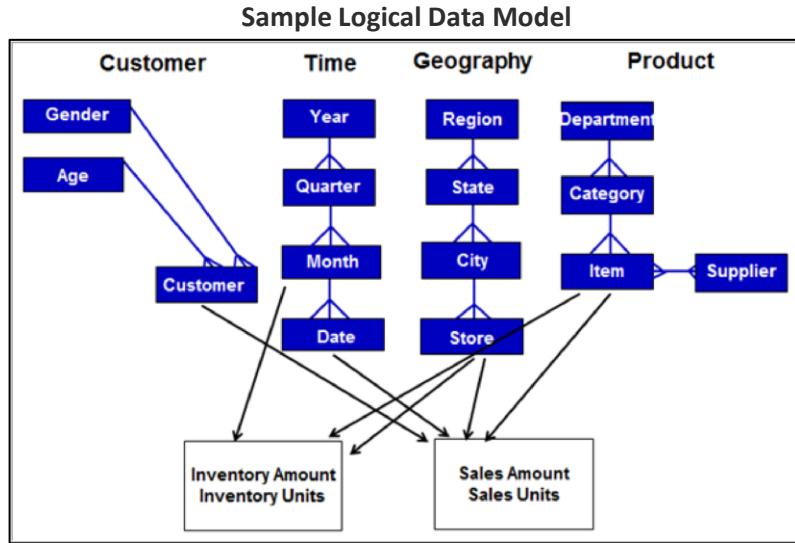
Architect allows multiple users to access schema objects simultaneously through two modes, Read Only and Edit.

- Read Only mode allows users to view schema object definitions without locking the schema from other users. Users in Read Only mode cannot make changes to schema objects. Additionally, users in Read Only mode cannot access schema editors that let you make updates to the project.
- Edit mode provides all capabilities of modifying schema objects and locks the schema object from being modified by all other users. Only one user can use Edit mode for a project at a given time. This control mechanism prevents schema inconsistencies.

Exercise solutions

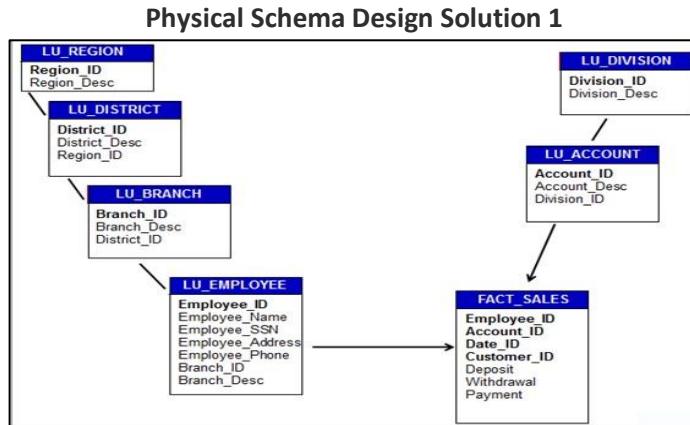
Exercise 1.1 solution: Designing the Logical Data Model

The following image shows one possible logical data model solution for the listed scenario:

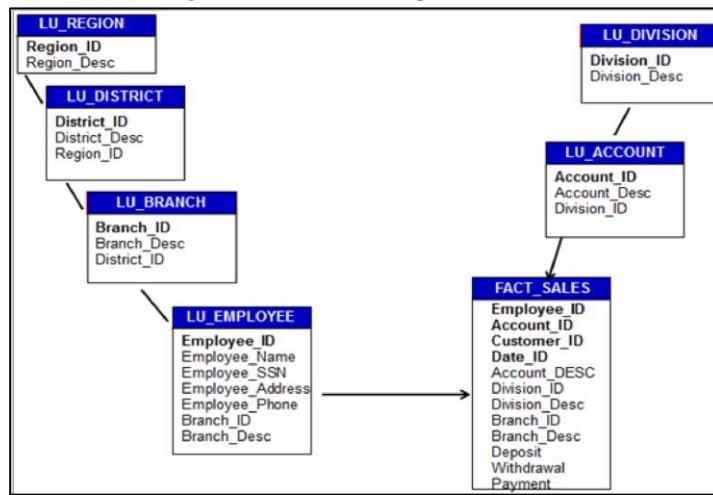


Exercise 1.2 solution: Designing the data warehouse schema

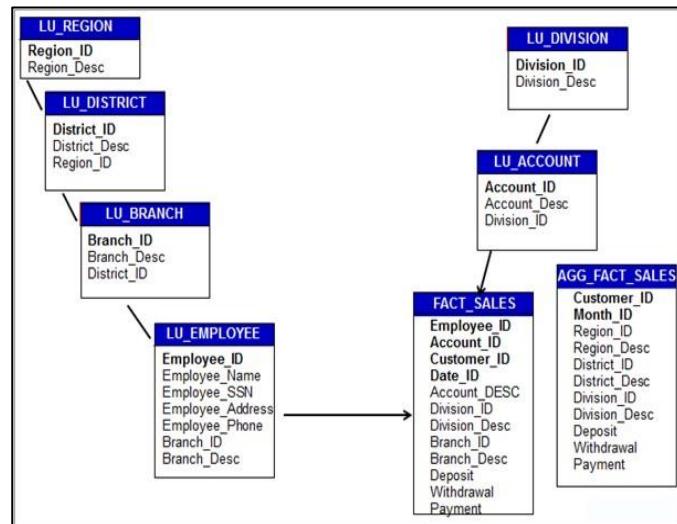
The following images show some of the possible solutions:



Physical Schema Design Solution 2



Physical Schema Design Solution 3



CHAPTER 2: WORKING WITH TABLES

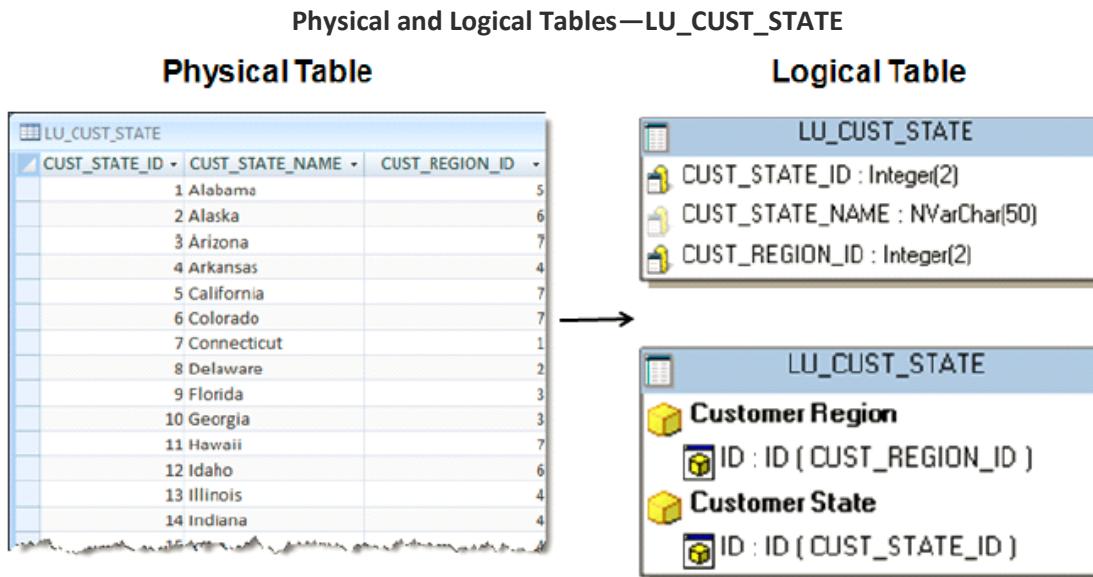
Project tables in MicroStrategy store information about the tables in your data warehouse. Project tables identify the subset of data warehouse tables that you are using in the MicroStrategy project. Your MicroStrategy project effectively ignores any data warehouse tables that are not added as project tables. The tables selected for the project are the tables users can query when executing reports.

Physical tables versus logical tables

When selecting a physical table from the data warehouse to include in a project, Architect automatically creates a corresponding logical table in the metadata. For example, if you pull the LU_CUSTOMER table from the data warehouse into the project, a corresponding LU_CUSTOMER table is automatically created in the metadata.

To obtain result sets, the MicroStrategy Engine executes report SQL against the physical tables that store the actual data. Logical tables store information about related physical tables, including table names, column names, data types, and schema objects associated with the table columns.

The following image shows the physical and logical tables for the LU_CUST_STATE table:



The LU_CUST_STATE physical table contains the CUST_STATE_ID, CUST_STATE_NAME, and CUST_REGION_ID columns. The corresponding LU_CUST_STATE logical table has both a physical and a logical view. The physical view displays the aforementioned columns and their data types. The logical view shows the attributes and attribute forms in the project that are mapped to the table columns.

Using layers when creating project tables

Layers are an organizational aid in Architect that enable you to group project tables based on the logical data model. This allows you to focus on only the tables needed to complete a particular task. For example, you can

create a layer that contains only the fact tables needed to create a related set of facts, or you can create a layer that contains only the lookup and relationship tables needed to create the attributes that belong to a particular hierarchy.

Although layers are not required, they allow you to organize your project tables into manageable chunks.

Viewing and modifying project table properties

All project tables have customizable settings like location, number of rows, mapped attributes, facts, columns, and so on. Settings can be changed in the Properties pane. The following table lists project table settings and their descriptions:

Category	Property	Description
Definition	ID	Globally Unique Identifier (GUID) that identifies the table in the metadata Note: You cannot modify this property.
	Name	Logical table name
	Description	Description of the logical table Note: This property does not contain a value if a description has not been entered.
	Hidden	Determines whether the logical table is a hidden object
	Location	Project folder location for the logical table Note: You cannot modify this property.
	Database Name	The name of the physical table that corresponds to the logical table
	Row Count	Number of rows in the physical table Note: This property does not contain a value unless you calculate the rows count for the table. You cannot modify this property.
	Table Name Space	Table name space that the physical table resides in Note: This property does not contain a value unless you use table name spaces in the data source. You cannot modify this property.

Logical Size	Logical size of the table as calculated by Architect Note: Changes to the attributes or facts that are mapped to a table can affect the logical table size. The updated logical table size is not displayed in the Properties pane until you update the project schema.
Logical Size Locked	Determines whether the logical size of a table is locked or recalculated by Architect during schema updates Note: You generally lock logical table size if you manually change the value to preserve this change during schema updates.
Primary DB Instance	Primary database instance for the table
Secondary DB Instances	Secondary database instances for the table Note: This property does not display unless the table is associated to more than one database instance.

Category	Property	Description
Mapped Attributes	<Attribute Name>	Column that maps to the attribute Note: Each attribute mapped to the table is listed as a separate property. This property does not display unless the table is mapped to an attribute.
Mapped Facts	<Fact Name>	Column that maps to the fact Note: Each fact mapped to the table is listed as a separate property. This property does not display unless the table is mapped to a fact.
Member Columns	<Column Name>	Data type of the column Note: Each attribute mapped to the table is listed as a separate property.

The Mapped Attributes and Mapped Facts categories display only if attributes or facts are mapped to a table.

To modify the properties of a project table, do one of the following:

- On the **Project Tables View** tab, select the table to modify properties for.
- In the Properties pane, click the **Tables** tab. In the drop-down list, select the table to modify properties for.

Now that you understand the objects that make up the MicroStrategy schema, let's explore the tool that you will use to create and manage those objects - MicroStrategy Architect.

Exercises

In this set of exercises, you will configure the Warehouse Catalog to load physical tables from the data warehouse, and then pull fact and lookup tables into the My Demo Project that you created in previous exercises. In subsequent exercises, you will use these tables to create facts and attributes.

Exercise 2.1: Configuring the Warehouse Catalog

To pull tables into your MicroStrategy project, you need to see a catalog of tables that are available in your data warehouse. In this exercise, you will configure the Warehouse Catalog to connect to your data warehouse and load the available tables.

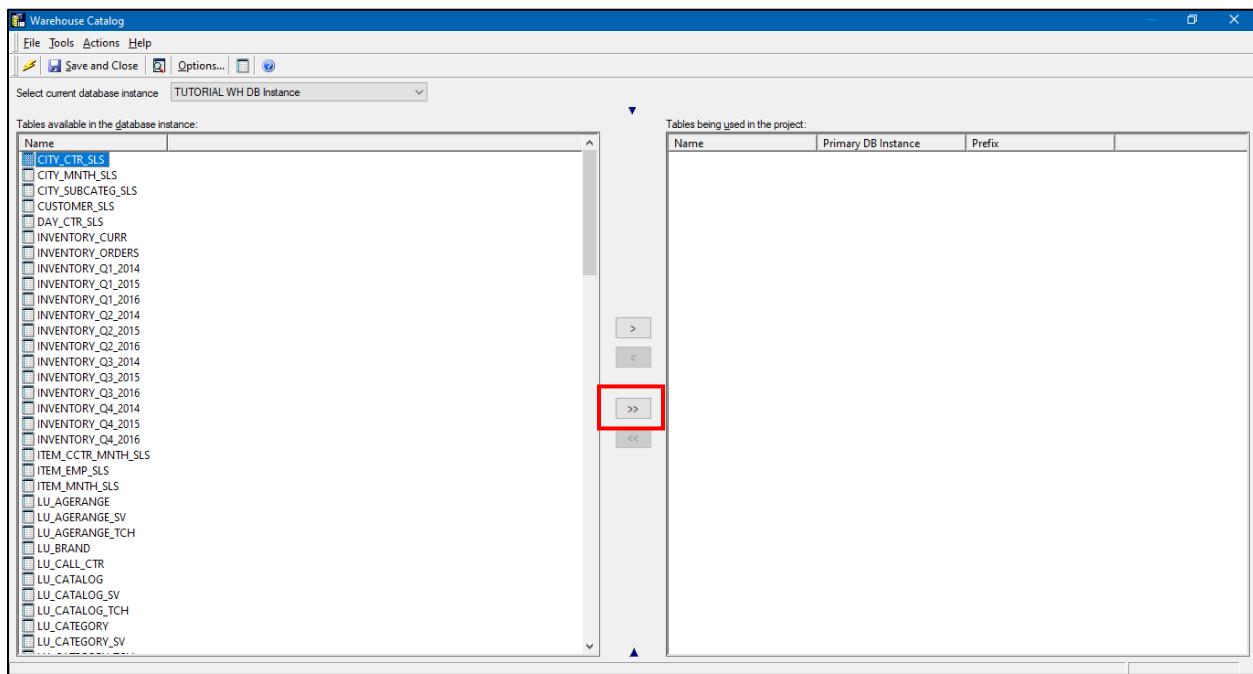
To load tables in the Catalog

1. In Developer, under the My Tutorial Project Source, double-click **My Demo Project** to expand the project.

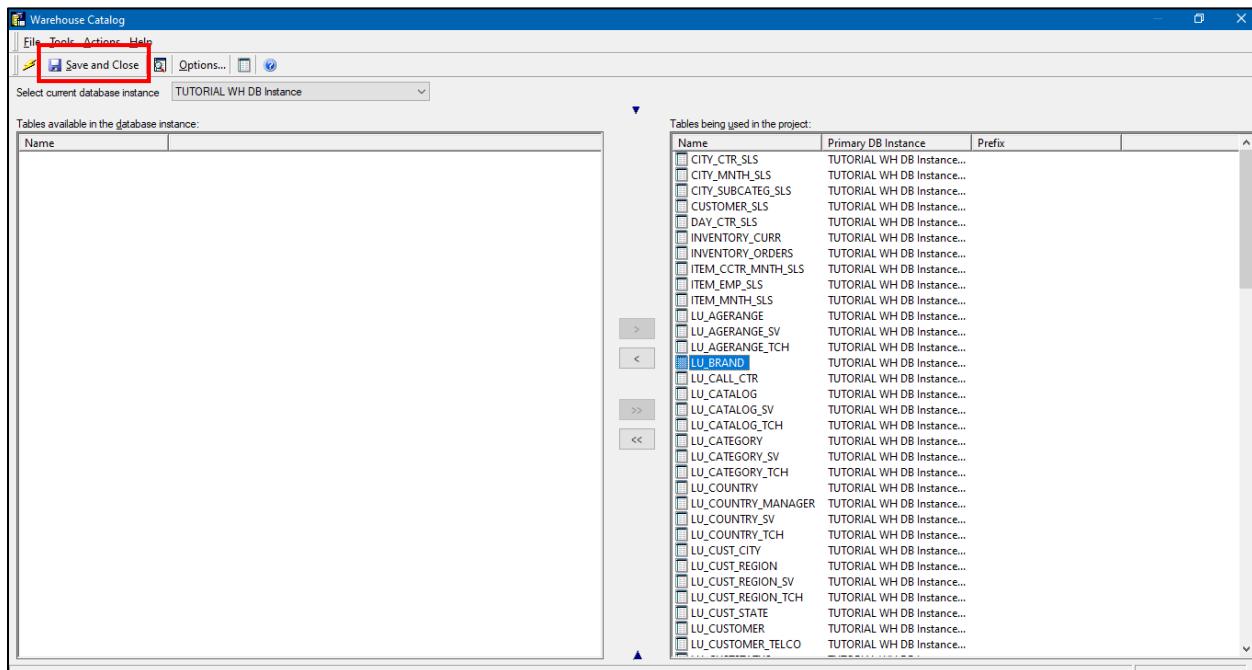
If you were logged out, you can log in to **My Tutorial Project Source** using the following credentials:

- login id: administrator
- password: (none)

2. From the **Schema** menu, select **Warehouse Catalog**.
3. In the Warehouse Database Instance window, in the drop-down list, select **TUTORIAL WH DB Instance** as the primary database instance for the project.
4. Click **OK**.
5. In the Warehouse Catalog Browser window, click **No** on the message regarding modifying the Warehouse Catalog Browser options.
6. Click the double right arrow to select all the tables.



7. Once you see the tables in the right pane, click **Save and Close**



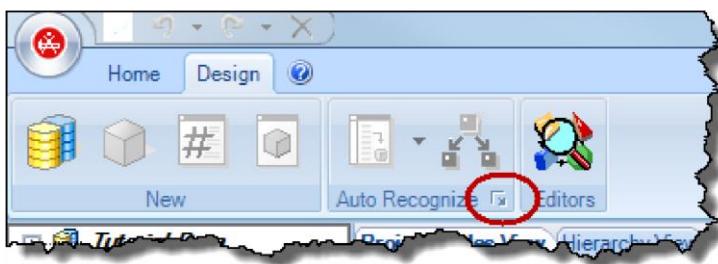
Exercise 2.2: Adding fact tables to the project

Fact tables store measurable information in your data warehouse. These tables will be used later to create fact schema objects in your MicroStrategy project.

In this exercise, you will use Architect to add fact tables from your data warehouse to My Demo Project.

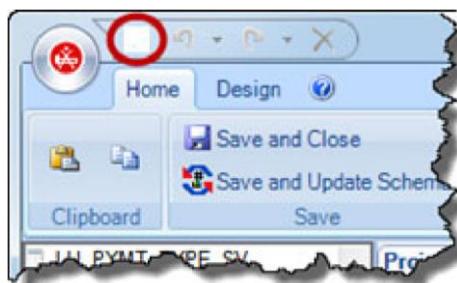
You will also disable automatic column recognition and relationship recognition in Architect, which are enabled by default. When these options are enabled, Architect automatically creates facts and attributes, and defines attribute relationships in your project when you pull in a table. When these options are disabled, you are required to manually create attributes and facts, and define attribute relationships.

Add fact tables to the project

1. In Developer, in the My Tutorial Project Source, in My Demo Project, from the **Schema** menu, select **Architect**.
2. If you were logged out, you can log in to My Tutorial Project Source using the login ID of **administrator**, and a blank password.
3. If the Read Only window is displayed, select **Edit: This will lock all schema objects in this project from other users** and click **OK**.
4. To disable automatic column recognition and relationship recognition:
 - a. In Architect, click the **Design** tab.
 - b. In the Auto Recognize area, click the arrow to open the Automatic Schema Recognition window.
 - c. In the Automatic Schema Recognition window, under Automatic column recognition, click **Do not auto recognize**.
 - d. Under Recognize Relationships, ensure that **Do not automatically create relations** is selected.
 - e. Click **OK**.
5. To enable automatic metric creation:
 - a. In Architect, click the **Architect** icon, in the top left corner.



- b. From the **Architect** menu, select **Settings**.
 - c. In the Architect Settings window, click the **Metric Creation** tab.
 - d. Select the **Sum** check box, if not already selected, then click **OK**.
6. To create a Fact Tables layer:
- a. In Architect, click the **Project Tables View** tab.
 - b. Click the **Home** tab at the top of the screen.
 - c. In the Layer area, click **Create New Layer**.
 - d. In the Architect window, type **Fact Tables** as the layer name.
 - e. Click **OK**.
7. To add fact tables to the project:
- a. In the Warehouse Tables pane on the left, expand **TUTORIAL WH DB Instance** to view all the tables.
 - b. Drag the following fact tables to the Fact Tables layer on the Project Tables View tab:
 - city_subcateg_sls
 - customer_sls
 - day_ctr_sls
 - order_detail
 - order_fact
 - c. Alternatively, you can add the desired tables to the All Project Tables layer, then right-click in the Fact Table layer and select **Edit Layer Element**. Select the tables to add to the Fact Table layer and click **OK**.
 - d. On the Home tab, in the Auto Arrange Table Layout section, click **Regular** to arrange the tables.
 - e. Click **Save**.



8. If a Change Comments window displays, click **Do not show this screen in the future** and click **OK**.

Exercise 2.3: Adding lookup tables to the project

Lookup tables in your data warehouse provide ID and descriptions for your attributes. To connect this information to MicroStrategy you create logical tables in Architect to store information about the lookup tables in your data warehouse.

In this exercise, you will add lookup tables to My Demo Project using Architect. To organize the logical tables, you will create the Geography and Time layers. Next, you will add the appropriate tables to each layer.

To add lookup tables to the project

1. In My Demo Project, on the Home tab, in the Layers area, click **Create New Layer**.

Ensure that you do not have any tables in the current layer selected before clicking **Create New Layer**. If you have tables selected, they are automatically included in the new layer.

2. In the MicroStrategy Architect window, type **Geography** as the layer name and click **OK**.
3. Repeat the steps above to create the **Time** layer.
4. To add lookup tables to the project:
 - a. In the **Layers** drop-down list, select the **Geography** layer.
 - b. Drag the following tables in the Warehouse Tables pane to the Geography layer on the Project Tables View tab:
 - lu_call_ctr
 - lu_country
 - lu_dist_ctr
 - lu_employee
 - lu_region

Alternatively, you can add the desired tables to the All Project Tables layer, then right-click in the Geography layer and select **Edit Layer Element**. Select the tables to add to the Geography layer and click **OK**.

- c. On the Home tab, in the Auto Arrange Table Layout section, click **Regular** to arrange the tables.
- d. In the **Layers** drop-down list, select the **Time** layer.

e. Drag the following tables in the Warehouse Tables pane to the Time layer on the Project Tables View tab:

- lu_day
- lu_month
- lu_month_of_year
- lu_quarter
- lu_year

Alternatively, you can add the desired tables to the All Project Tables layer, then right-click in the Time layer and select **Edit Layer Element**. Select the tables to add to the Time layer and click **OK**.

- f. On the Home tab, in the Auto Arrange Table Layout area, click **Regular** to arrange the tables.
5. To save and update the project schema, on the Home tab, in the Save area, click **Save and Close**.
6. In the Schema Update window, ensure the following check boxes are selected:
- Update schema logical information
 - Recalculate table keys and fact entry levels
 - Recalculate table logical sizes
7. Click **Update**.

If the Change Comments window displays when you update the schema, select the **Do not show this screen in the future** check box and click **OK**.

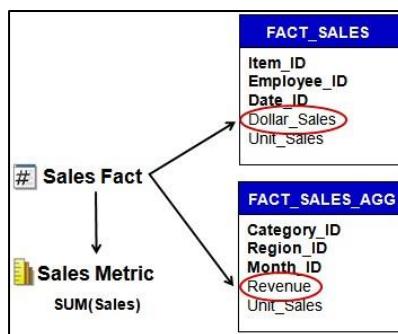
CHAPTER 3: WORKING WITH FACTS

Facts are logical objects that relate to measurable data in your data warehouse. Analysts can aggregate facts to create metrics that are displayed on MicroStrategy reports. Facts point to physical columns in the data warehouse, while metrics perform aggregations on those columns.

Now that you have pulled your fact tables into the MicroStrategy project, you can create fact schema objects. Create your facts based on the logical data model, and map them to the appropriate columns in the data warehouse schema.

Facts create a layer of abstraction between the underlying structure of your data warehouse and the metrics users want to see on reports. Consider the following example:

Sales Fact Layer of Abstraction



In the data warehouse, Sales information exists in both the FACT_SALES and FACT_SALES_ AGG tables. Sales data is stored in these tables using different column names—Dollar_Sales and Revenue.

If a report returns Sales data, it can either aggregate the Dollar_Sales column or the Revenue column. The query targets the appropriate table based on the attributes used in the report. To enable the MicroStrategy Engine to use a single fact to access either table, the Sales fact must be mapped to both tables.

If the Sales fact is not mapped to both tables, analysts would need to create two distinct metrics—one defined as SUM(Dollar_Sales), and another defined as SUM(Revenue). The analysts would then need to choose the appropriate metric to use on each report, based on the attributes present.

The Sales fact creates a layer of abstraction that obscures column mapping for end users. If you map a fact correctly, analysts can create metrics that seamlessly connect to the desired information in the data warehouse.

The layer of abstraction created by facts provides the following benefits:

- Report designers and end users do not need to understand the structure of the data warehouse.
- The data warehouse can contain fact columns with different names that store the same data.
- You do not have to resolve column naming discrepancies in the data warehouse.

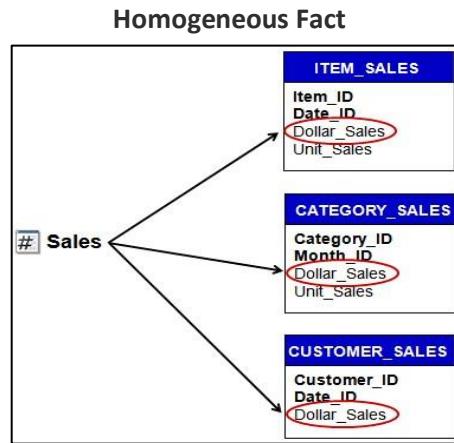
Types of facts

The facts you create in MicroStrategy are associated with columns in your data warehouse tables. The same fact can be mapped to any number of tables. Depending on the structure of your data warehouse, your facts will either be homogeneous or heterogeneous.

Homogeneous facts

A homogeneous fact points to the same column or set of columns in every table to which it is mapped.

The following example shows a homogeneous fact:

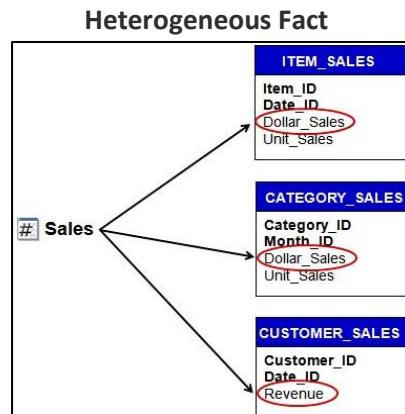


The Sales fact maps to three different tables—ITEM_SALES, CATEGORY_SALES, and CUSTOMER_SALES. Sales is considered a homogeneous fact because it maps to the same Dollar_Sales column in each table.

Heterogeneous facts

While a homogeneous fact always maps to the same column or set of columns, a heterogeneous fact points to two or more columns in the tables to which it maps.

The following example shows a heterogeneous fact:



The Sales fact maps to three different tables—ITEM_SALES, CATEGORY_SALES, and CUSTOMER_SALES. In the first two tables, it maps to the Dollar_Sales column, but in the third table, it maps to the Revenue column. Sales is a heterogeneous fact because it maps to two different columns.

Types of fact expressions

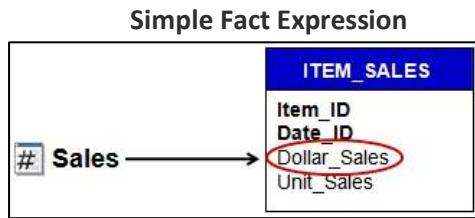
A fact expression consists of a column or set of columns to which the fact maps. All facts have at least one expression. However, facts can have any number of expressions. There are two primary types of fact expressions:

- Simple
- Derived

Simple fact expressions

A simple fact expression maps directly to a single fact column in any number of tables.

The following example shows a simple fact expression:



The Sales fact maps directly to the Dollar_Sales column in the ITEM_SALES table, creating a simple fact expression.

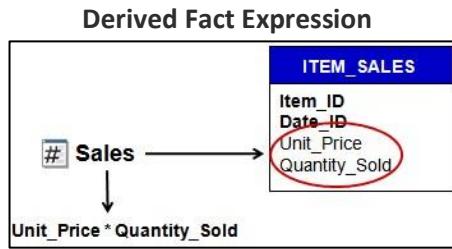
Derived fact expressions

A derived fact expression maps to an expression that calculates the fact values. A derived fact expression can contain multiple fact columns from the same table, mathematical operators, numeric constants, and various functions.

If you want to combine fact columns from different database tables in a derived fact expression, you create a logical view.

MicroStrategy provides a variety of functions that you can use to define fact expressions, including pass-through functions that enable you to pass SQL statements directly to the data warehouse.

The following example shows a derived fact expression:



The Sales fact maps to an expression that multiplies the Unit_Price and Quantity_Sold columns from the ITEM_SALES table.

MicroStrategy Architect enables you to create derived fact expressions at the application level, but you can alternatively store derived fact columns at the database level. Storing derived fact columns in the data warehouse allows calculations to be performed ahead of time during the ETL process. This method translates into simpler report SQL and better performance. If you implement a derived fact expression at the application level, the calculation is performed each time you process a query that uses the fact. The method you choose to handle derived facts depends on your ETL process and the performance impact of calculating derived fact expressions on the fly.

Creating facts

Architect provides multiple options to help you create facts. The optimal method to use depends on your project environment and the characteristics of individual facts. The following sections describe various fact creation methods and when to use them.

Using layers to organize fact tables

Before you start creating facts, organize the fact tables in your project into distinct layers. This process allows you to focus your attention on one category of facts at a time. Depending on the scale of your project, you may be able to group all of your fact tables in a single layer. In larger projects, you may create distinct layers for each fact category. Use your own judgment to organize your fact tables into logical layers that will help you create facts.

Fact creation methods

You can create facts using two different methods in Architect:

- **Manual fact creation** allows you to define each individual fact by mapping to columns in your data warehouse. This method gives you precise control over your project's facts and their definitions. Consequently, this process can be labor intensive.
- **Automatic fact creation** allows Architect to automatically identify and create facts. Through a set of heuristics, Architect uses automatic column recognition to identify columns in your data warehouse that are candidates for facts. After the facts are created, you can modify them as needed. This method provides a quick and easy way to reduce your fact creation effort, but you do need to verify that the

facts created by Architect meet your needs. When the automatic fact creation process is completed, make sure that the list of facts meets your expectations, and that the definition for each fact is correct.

The method you employ in your project depends on the physical structure of the tables and columns in the data warehouse and your user requirements. You can save time with automatic fact creation, but you must ensure that the resulting facts are accurate. Manual fact creation can be a more suitable option for projects that include complex fact requirements.

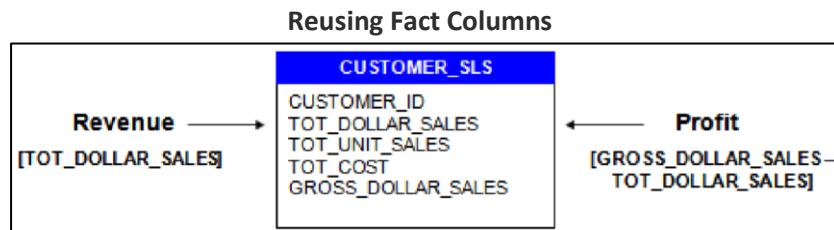
The remainder of this lesson outlines the manual fact creation process. Automatic fact creation and automatic column recognition are covered later in this course.

Creating facts manually

Manual fact creation allows you to exert full control over the facts in your project. Use Architect to create simple or derived fact expressions. You can also create heterogeneous facts that have multiple expressions.

Reusing fact columns

In Architect, after you map a fact to a given column, the table no longer displays that column name. This allows you to easily track the columns that have not yet been used to create facts. However, in some cases, you may need to reuse a column in an expression for another fact. For example, consider the following scenario:



In this example, two facts use the **TOT_DOLLAR_SALES** column in their definitions. If you create the **Profit** fact first, you use both the **TOT_DOLLAR_SALES** and **GROSS_DOLLAR_SALES** columns in its expression. As a result, these columns would no longer appear in the table.

To allow column reuse, Architect provides an alternate method to access and select all table columns even if they have already been mapped. To use this fact creation method in Architect, right-click the table header and select **Create Fact**. In the Create New Fact Expression window, all columns are displayed in the Available Columns list, including the columns that have been used to create other facts.

Exercise 3.1: Creating facts

Facts are logical objects that relate to measurable information in your data warehouse. Analyst aggregate facts to create the metrics they display on reports.

In this exercise, you will use Architect to manually create the following facts in My Demo Project:

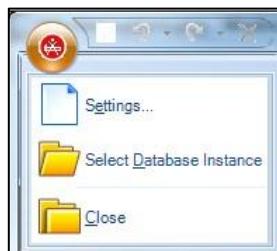
Expression	Fact Name
DISCOUNT	Discount

TOT_COST	Cost
TOT_DOLLAR_SALES	Revenue
UNIT_COST	Unit Cost
UNIT_PRICE	Unit Price
TOT_UNIT_SALES	Units Sold

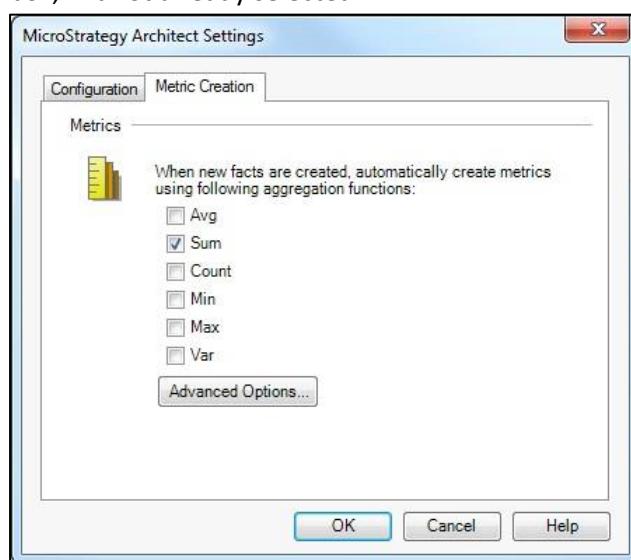
When you create a new fact in Architect, a corresponding metric is automatically created for you. These first few steps ensure that the Sum aggregation function is used to create the metrics. You can select other aggregation functions if your project needs them.

To set the metric aggregation function

1. From the **Architect** menu, click **Settings**.



2. Click the **Metric Creation** tab.
3. Select the **Sum** check box, if it not already selected.



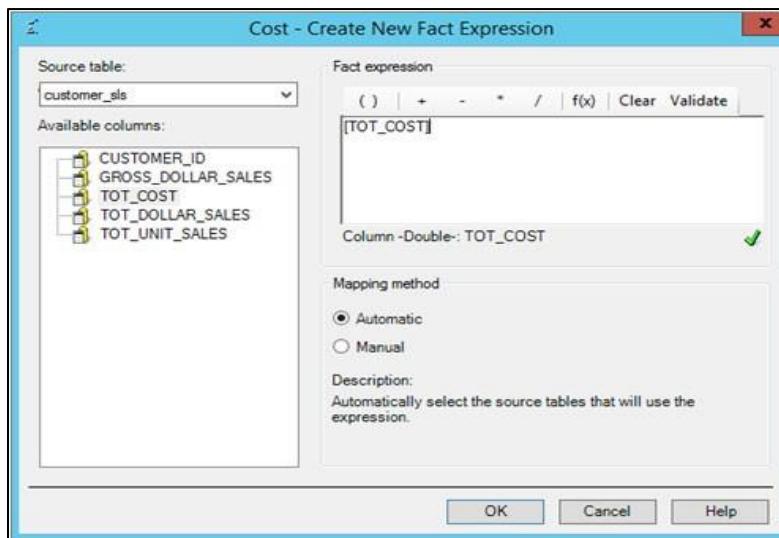
4. Click **OK**.

Create facts

1. In MicroStrategy Developer, in the My Demo Project, on the **Schema** menu, click **Architect**.
2. *If you were logged out, you can log in to My Tutorial Project Source using the **administrator** user's login credentials.*
3. In Architect, click the **Project Tables View** tab.
4. On the Home tab, in the Layer section, in the **Layers** drop-down list, select the **Fact Tables** layer.

Create a fact

5. On the Project Tables View tab, right-click the header of any table that contains TOT_COST and select **Create Fact**.
6. In the MicroStrategy Architect window, in the box, type **Cost** as the fact name.
7. Click **OK**.
8. In the Create New Fact Expression window, double-click **TOT_COST** to define the fact expression.
9. Under **Mapping method**, ensure **Automatic** is selected.



10. Click **OK**. The Cost fact is created in My Demo Project.

11. Repeat the steps to create the following facts:

Column Name	Table Name	Fact Name
DISCOUNT	ORDER_DETAIL	Discount
TOT_DOLLAR_SALES	This column is available in most of the tables; you can use CITY_SUBCATEG_SLS	Revenue
UNIT_COST	ORDER_DETAIL	Unit Cost
UNIT_PRICE	ORDER_DETAIL	Unit Price
TOT_UNIT_SALES	This column is available in most of the tables; you can use CITY_SUBCATEG_SLS	Units Sold

*Alternatively, you can create these facts by right-clicking the appropriate column in the table and selecting **Create Facts**. Because this method defaults the fact name to the mapped column name, make sure you rename the facts when appropriate. To rename a fact, right-click it and select **Rename**. In the Architect window, in the box, type the fact name, and click **OK**.*

12. On the **Home** tab, in the Save area, click **Save and Close**.
13. **Update the schema**
14. In the Update Schema window, ensure the following check boxes are selected:
 - a. Update schema logical information
 - b. Recalculate table keys and fact entry levels
 - c. Recalculate table logical sizes
15. Click **Update**.

CHAPTER 4: WORKING WITH ATTRIBUTES AND RELATIONSHIPS

Attributes provide business context and help you answer questions about your measurable fact data. Analysts use attributes to describe facts at various levels of detail on reports and documents. For example, the following image shows two reports with the same metric aggregated to different levels, based on the attributes used on the reports:

Attributes and Metric Aggregation				
		Metrics	Profit	
Year	Category			
2010	Books	\$139,952		
	Electronics	\$1,057,330		
	Movies	\$62,644		
	Music	\$44,215		
2011	Books	\$187,027		
	Electronics	\$1,410,402		
	Movies	\$83,637		
	Music	\$59,019		
2012	Books	\$242,299		
	Electronics	\$1,821,871		
	Movies	\$108,417		
	Music	\$76,811		
Total		\$5,293,624		

Attributes and Metric Aggregation				
		Metrics	Profit	
Region	Call Center			
Central	Milwaukee	\$637,545		
	Fargo	\$126,778		
	Washington, DC	\$473,200		
	Charleston	\$199,884		
Northeast	Boston	\$224,495		
	New York	\$1,076,237		
	San Francisco	\$156,330		
	Seattle	\$110,655		
Northwest	New Orleans	\$504,990		
	Memphis	\$301,966		
	Atlanta	\$157,963		
	Miami	\$178,713		
South	San Diego	\$449,553		
	Salt Lake City	\$111,779		
	Web	\$583,538		
	Total	\$5,293,624		

The total for the Profit metric on each report is the same. However, the first report displays profit by Year and Category because those are the attributes on the report and are not directly related. The second report displays profit by Call Center because the Region and Call Center attributes on the template are directly related and call centers represent the lowest level of detail.

Analysts can also use attributes to directly define the level of aggregation for a specific metric. When used in filters, attributes qualify the result set of a report based on attribute data. For example, the following image shows two reports with the same template that return different result sets based on the attributes in their respective filters:

Attributes and Qualification				
Report details		View filter		
Report Description:				
Report Filter: Category = Books, Movies				
Report details		View filter		
Report Description:				
Report Filter: Year = 2012				
Report details		View filter		
Report Description:				
Report Filter: Year = 2012				
Year	Category	Metrics	Profit	
2010	Books	\$139,952		
	Movies	\$62,644		
2011	Books	\$187,027		
	Movies	\$83,637		
2012	Books	\$242,299		
	Movies	\$108,417		

The attributes displayed on each report are the same, but the filters are different. The first report displays profit for all years for the Books and Movies categories because those Category elements are specified in the filter. The second report displays profit for all categories for 2012 only because that Year element is used in the filter.

Now that you have created facts in your MicroStrategy project, you are ready to create attributes to provide context for those facts. You create attributes based on your logical data model, and map them to columns in the data warehouse physical schema. To define how attributes work together, you will define their relationships. Once this process is complete, analysts will be able to use attributes as components in reports, filters, metrics, and so on.

About attribute forms

An attribute form contains supplementary identification and description information about an attribute. Using forms, an attribute can be represented in various ways. For example, a Customer attribute may have an identification number, name, address, and birthday forms.

Attribute forms enable analysts to display different types of descriptive information about an attribute on reports. For example, an ID form may be a unique customer identification number, while a description form may be a customer name, address, email, and so on.

Attribute forms map directly to the columns in the data warehouse to retrieve ID and description information. All attributes have an ID form, and most have at least one description form. For example, you may store the following information about customers in the data warehouse:

Customer Information	
LU_CUSTOMER	
Customer_ID	
Customer_Name	
Home_Phone	
Home_Address	
Email	
Birth_Date	
Gender_ID	
Income_ID	

The LU_CUSTOMER table contains a variety of information about each customer that may be useful to view or analyze in the reporting environment. When you create your attribute forms in your project, consider how users plan to use attributes in reports, and whether the unique characteristics of attribute forms will meet those needs.

Attribute form relationships

Attribute forms must have a one-to-one relationship with other forms that are part of the same attribute. For example, if Name and Email are attribute forms for Customer, a customer cannot have more than one name or more than one email address.

If customers need to have more than one email address, then the relationship between Email and Customer is not one-to-one. In this case, if you want to display all the email addresses for a single customer, you would create a separate Email attribute instead of creating it as a Customer attribute form.

If you create an attribute form that has a one-to-many relationship with the attribute it describes, only the first element of that attribute form will display on reports.

User interaction with attribute forms

The ability to create multiple forms for an attribute provides flexibility for analysts who create and interact with reports. When you create multiple attribute forms, analysts can display an attribute in various ways on different reports.

The following image shows a report that displays multiple attribute forms for the Customer attribute:

Customer Attribute Forms					Metrics	Revenue
Customer	ID	Last Name	First Name	Address	Email	
7796	Aaby	Alen	864 Kalispell Dr	aaaby54@yahoo.demo	\$3,104	
4459	Aasen	Beatrice	242 E Water St	baasen20@hotmail.demo	\$4,022	
1210	Aba-Bulgú	Leslie	1402 S Desert Vista Dr	laba-bulgú05@aol.demo	\$2,196	
3888	Abad	Bekir	677 Edgemere Ave	babad22@free.demo	\$2,204	
4602	Abbenhaus	Lonzie	7310 E Black Rock Rd	labbenhaus41@aol.demo	\$3,639	
6211	Abbott	Rosella	RR 1	rabbott56@yahoo.demo	\$5,513	
5834	Abdyusheva	Dominique	2395 Grace Chapel Rd	dabdyusheva45@free.demo	\$1,563	
4238	Abel	Edward	705 Bayard Rd	eabel79@free.demo	\$1,310	
8996	Abeleda	Devon	Hidden Acres Rd	dabeleda95@univ.demo	\$1,241	
9835	Abeles	Doyle	6 Berry Pl	dabeles51@yahoo.demo	\$233	
1711	Aberbach	Henning	Cutsinger St	haberbach27@yahoo.demo	\$4,850	
3958	Abitia	Nokeo	2312 Lincolnwood Dr	nabitia65@univ.demo	\$4,914	
2864	Abney	Heidi	Fisherman Dr	habney54@yahoo.demo	\$1,514	
2398	Abogado	Adrian	4525 Murrays Run Rd	aabogado41@univ.demo	\$3,564	
6463	Abogado	Gabriela	874 Judkin Mill Rd	gabogado14@free.demo	\$2,494	
8828	Abou-Arabi	Roy	30 Peach Tree Dr	rabou-arabi63@free.demo	\$5,017	
3118	Abraham	Dori	1547 Josephine St	dabraham06@aol.demo	\$2,801	

This report displays each customer's ID, last name, first name, address, and email under the Customer attribute header.

Analysts can use attribute forms to achieve the following reporting behaviour:

- **Display:** Display different forms of an attribute on reports or in the Data Explorer when users browse the attribute.
- **Sorting:** Sort report data using attribute forms. Sort using any attribute form, not just those displayed on a report.
- **Qualification:** Qualify data based on the elements of any attribute form. Qualify using any attribute form, not just those displayed on a report.

If analysts simply need to display, sort, or qualify descriptive data, an attribute form can fulfill their needs. However, if analysts need to aggregate metrics based on descriptive data, you must create a new attribute.

For example, analysts probably do not need to aggregate revenue data for customers based on their name, home address, home phone, email, or birth date. These descriptors can be created as attribute forms of the Customer attribute. On the other hand, analysts may want to aggregate revenue data based on customer gender or income. To make this aggregation possible in reports, you would create Gender and Income as distinct attributes.

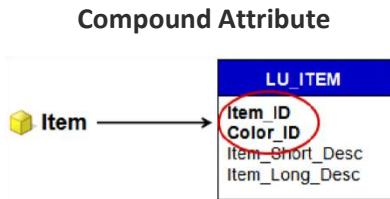
Types of attributes

Each attribute's type depends on how its data is stored in the data warehouse. Depending on the physical structure of your data, you will create compound, homogeneous, and heterogeneous attributes.

Compound attributes

A compound attribute maps to two or more columns to generate its ID form. If an attribute uses a compound primary key in the data warehouse to uniquely identify its elements, it is considered a compound attribute.

The following example shows a compound attribute:

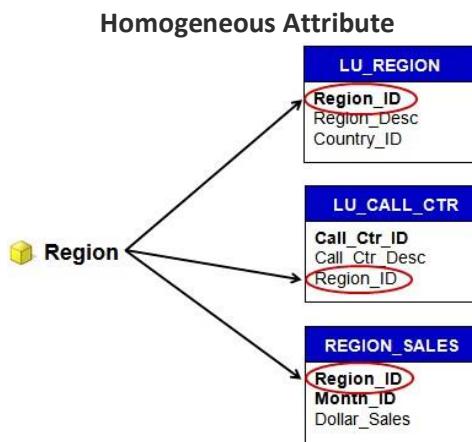


To create its primary key, the Item attribute uses two different columns in the LU_ITEM table—Item_ID and Color_ID. To create an ID form for this compound attribute, you must map the form to these two columns.

Homogeneous attributes

A homogeneous attribute form points to the same column or set of columns in its mapped tables.

The following example shows a homogeneous attribute:

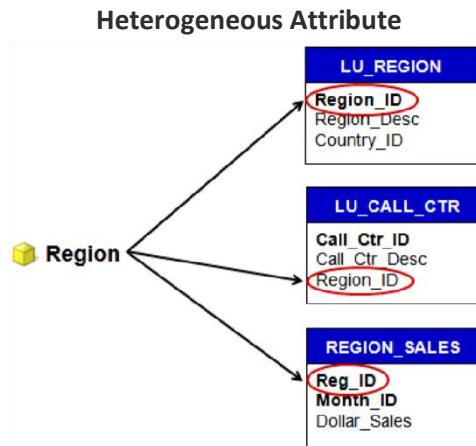


The ID form for the Region attribute maps to three different tables—LU_REGION, LU_CALL_CTR, and REGION_SALES. Region is considered a homogeneous attribute because its ID form maps to the same Region_ID column in each table.

Heterogeneous attributes

A heterogeneous attribute contains at least one attribute form that points to two or more columns in its mapped tables.

The following example shows a heterogeneous attribute:



Region is considered a heterogeneous attribute because its ID form maps to two different columns. The ID form for the Region attribute maps to three different tables—LU_REGION, LU_CALL_CTR, and REGION_SALES. The ID form maps to the Region_ID column in the first two tables, and to the Reg_ID column in the last table.

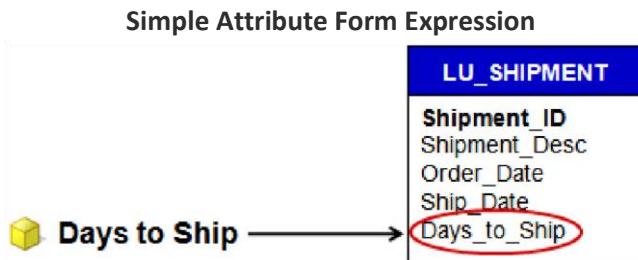
Types of attribute form expressions

An attribute form expression identifies the database columns and functions required to retrieve information from the data warehouse. All attribute forms have at least one expression. The attribute form expressions you create are either simple or derived.

Simple attribute form expressions

A simple attribute form expression maps directly to a single attribute column. The expression can map to the same column in any number of tables.

The following example shows a simple attribute form expression:



The ID form for the Days to Ship attribute maps directly to the Days_to_Ship column in the LU_SHIPMENT table. This is a simple attribute form expression because a single column is used and no operations are implemented.

Derived attribute form expressions

A derived attribute form maps to an expression that calculates its values. The expression can contain multiple attribute columns from the same table, mathematical operators, constants, and various other functions.

The following example shows a derived attribute form expression:



The Days to Ship attribute form uses an expression to calculate its values. This expression subtracts the Order_Date from the Ship_Date to derive the number of days it takes to ship the order.

MicroStrategy provides a variety of out-of-the-box functions that you can use to define attribute form expressions, including pass-through functions that enable you to pass SQL statements directly to the data warehouse.

If you want to combine attribute columns from different database tables in a derived attribute form expression, you must create a logical view. For detailed information on logical views, see the *MicroStrategy Advanced Data Warehousing* course.

A derived attribute form expression in MicroStrategy triggers a calculation each time the form is used in a query. If you want to increase the performance of these queries, you can forgo the derived expression in MicroStrategy in favor of a derived attribute stored at the database level. This option allows you to perform the required calculation during the ETL process and store the result in a single column.

Creating attributes in Architect

Now that you understand attribute types and expressions, you can use Architect to seamlessly create attributes in your project. Reference the steps in the following section to build various types of attributes and establish their relationships.

Grouping attributes using layers

A layer is an organizational tool that helps you manage related groups of logical tables in Architect. To help you create attributes, you can organize relationship and lookup tables related to a specific hierarchy into a single layer. This strategy aligns the layers to help you focus only on pertinent tables as you create a group of related attributes.

Once you've established your layers in Architect, you can use the following methods to create attributes.

About manual and automatic attribute creation

Depending on the complexity of your data warehouse and your understanding of its structure, use one of the following methods to create attributes in Architect:

- **Manual** attribute creation requires you to create each individual attribute and corresponding attribute forms.

- **Automatic** attribute creation leverages Architect's algorithms to identify and create attributes and attribute forms in your project. After they are created, you must verify each attribute for accuracy and completeness.

Alternatively, you can use the Attribute Creation Wizard as a step-by-step guide to help you create multiple attributes.

Manually defining attribute details

To create attributes manually, you must map each attribute to one or more column in your data warehouse. You can use one of two methods to manually create attributes:

- Map an attribute to the same column in all project tables
- Map an attribute to a column only in specified project tables

Create attributes by initially mapping the ID form, and then add other forms as needed.

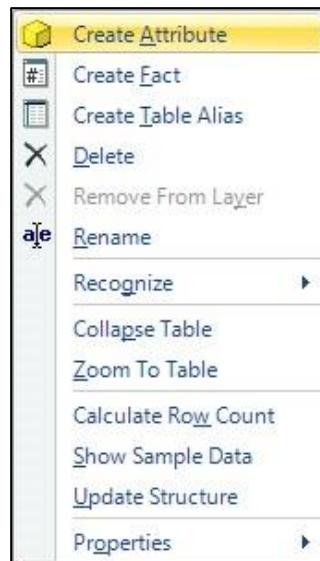
The steps below are not to be performed as an in-class exercise. They are for reference only.

Create an attribute with a simple form expression

1. On the Project Tables View tab, find the project table that you want to use as the primary lookup table for the attribute.

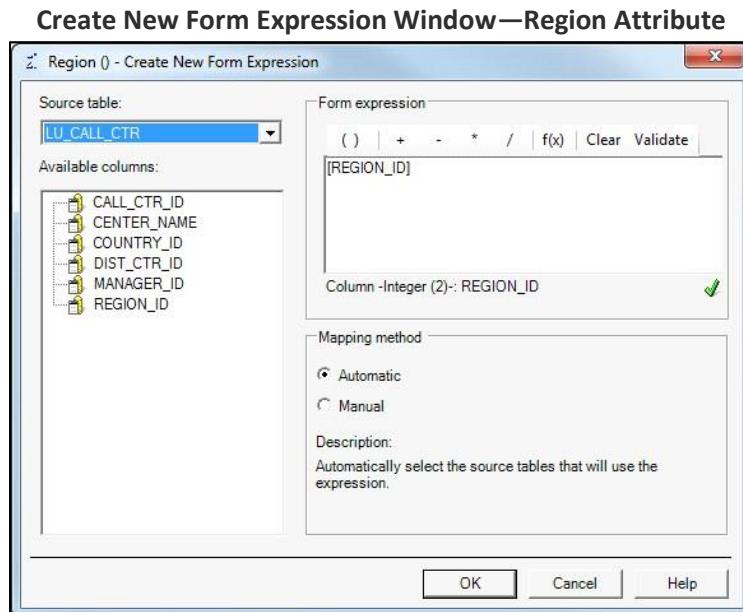
Architect automatically designates the table you use to create the attribute as the primary lookup table. You can change the primary lookup table at a later point.

2. Right-click the header of the project table and select **Create Attribute**.



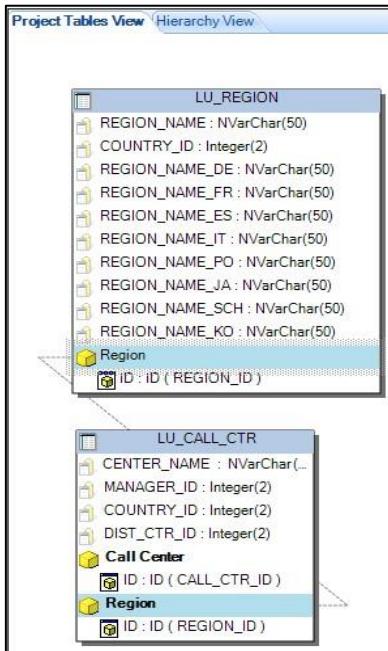
3. In the MicroStrategy Architect window, in the box, type a name for the attribute, then click **OK**.
4. In the Create New Form Expression window, define the ID form expression, then click **OK**.
5. To ensure the available columns in the table are visible, right-click the table, point to **Properties**, point to **Logical View**, and select **Display Available Columns**.

The following image shows the creation of the Region ID form in the Create New Form Expression window using the automatic mapping method:

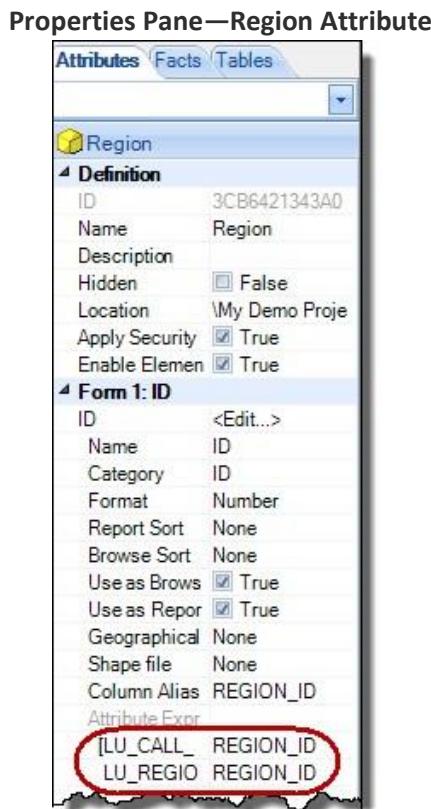


The following image shows the Region attribute defined for the REGION_ID column in the LU_REGION and LU_CALL_CTR tables:

Project Tables View Tab—Region Attribute



The following image shows the Region attribute displayed in the Properties pane:



The ID form for the Region attribute maps to the REGION_ID column in both the LU_REGION and LU_CALL_CTR lookup tables.

The steps below are not to be performed as an in-class exercise. They are for reference only.

Create an attribute with a simple form expression and map it to a column only in specified project tables

1. On the Project Tables View tab, right-click the header of the project table to be used as the primary lookup table for the attribute and select **Create Attribute**.

Architect automatically designates the table you use to create the attribute as the primary lookup table. You can change the primary lookup table at a later time.

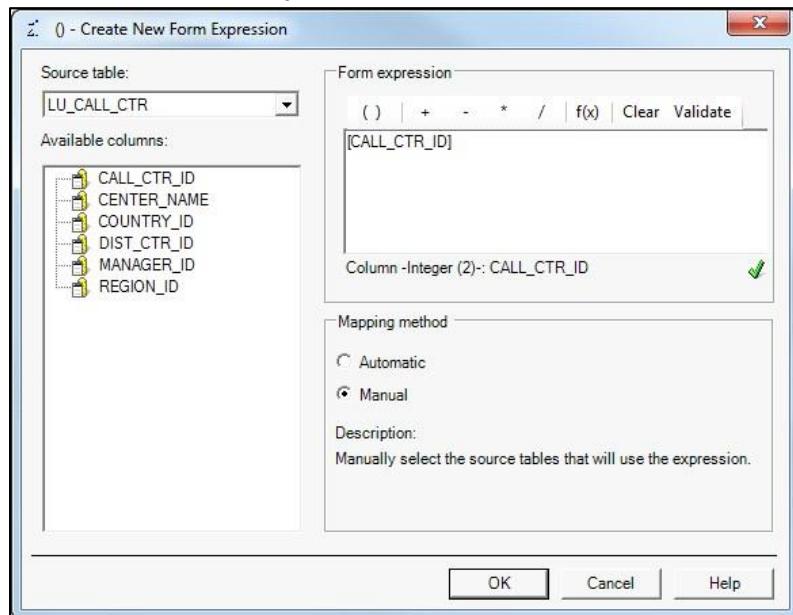
2. In the MicroStrategy Architect window, in the box, type a name for the attribute, then click **OK**.
3. In the Create New Form Expression window, define the ID form expression using the desired column.
4. Under **Mapping method**, click **Manual**. Then click **OK**.

This action creates an attribute with an ID form that maps to that column only for the selected project table. The new attribute automatically displays in the Properties pane.

5. To map the ID form for the attribute to additional tables, in the Properties pane, under the **Form 1: ID** category, select the **ID property**.
6. In the Properties pane, next to **Edit**, click ... (the **Browse** button).
7. In the Modify Attribute Form window, on the Definition tab, in the **Source tables** list, select the tables to map the ID form to. Click **OK**.

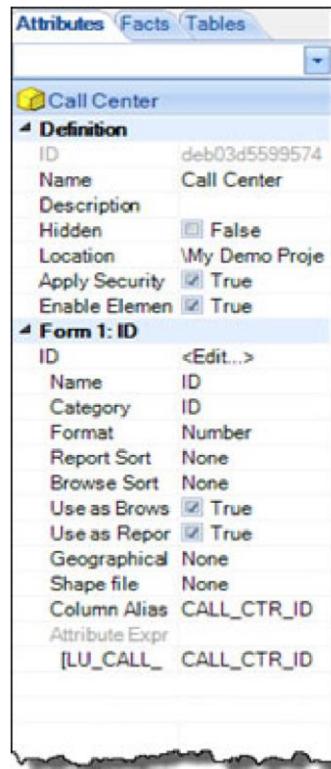
The following image shows the creation of the ID form for the Call Center attribute in the Create New Form Expression window using the manual mapping method:

Create New Form Expression Window—Call Center Attribute



The following image shows the Call Center attribute displayed in the Properties pane:

Properties Pane—Call Center Attribute



The ID form for the Call Center attribute maps to the CALL_CTR_ID column only in the LU_CALL_CTR lookup table. Other tables that include this column are not part of the attribute form definition.

Creating a derived attribute form expression

A derived attribute expression uses a function in conjunction with data warehouse columns to calculate attribute form values.

The steps below are not to be performed as an in-class exercise. They are for reference only.

Create a derived attribute form expression

1. On the Project Tables View tab, right-click the header of the project table you want to use as the primary lookup table for the attribute and select **Create Attribute**.

Architect automatically designates the table you use to create the attribute as the primary lookup table. You can change the primary lookup table at a later time.

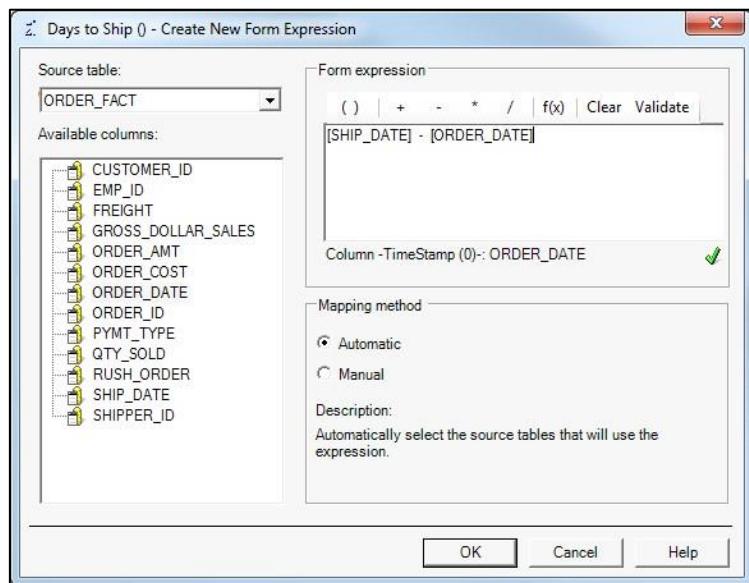
2. In the MicroStrategy Architect window, in the box, type a name for the attribute.
3. In the Create New Form Expression window, define the ID form expression using the desired columns and functions.

Use the toolbar above the box to insert parentheses, mathematical operators, or other functions. Information can also be typed directly in the box.

4. Under **Mapping method**, click **Automatic**. Then click **OK**.

The following image shows the Create New Form Expression window with a derived expression for the Days to Ship attribute:

Create New Form Expression Window—Days to Ship Attribute with Derived Form Expression



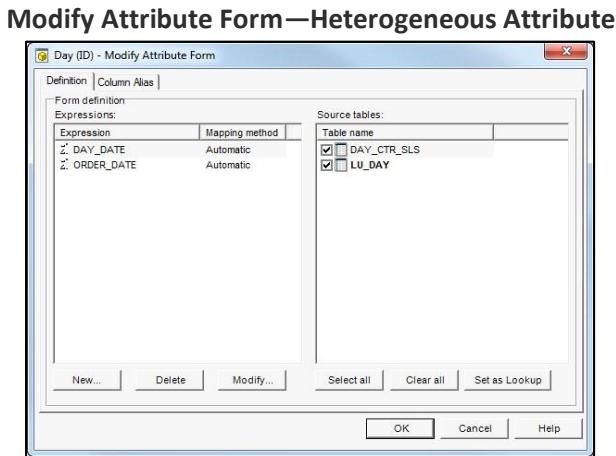
Creating a heterogeneous attribute

A heterogeneous attribute contains at least one attribute form that points to two or more columns in its mapped tables.

To create a heterogeneous attribute

1. On the Project Tables View tab, right-click the header of the project table you want to use as the primary lookup table for the attribute and select **Create Attribute**.
2. In the MicroStrategy Architect window, in the box, type a name for the attribute.
3. In the Create New Form Expression window, define the ID form expression as desired, then click **OK**.
4. On the Project Tables View tab, right-click the attribute form you just created and select **Edit**.
5. In the Modify Form Expression window, click **OK** to return to the Modify Attribute Form window.
6. In the Modify Attribute Form window, on the Definition tab, click **New**.
7. In the Create New Form Expression window, in the **Source table** list, select the source table for the new expression.
8. In the **Form expression** box, define the new form expression. Then click **OK**.
9. Click **OK** to close the Modify Attribute Form window.

The following image shows the Modify Attribute form with multiple expressions for the ID form of the heterogeneous Day attribute:



Creating a compound attribute

A compound attribute maps to two or more columns to generate its ID form.

In Architect, create a compound attribute by assigning two or more attribute forms to the ID form category. When more than one attribute form has been assigned to the same form category, a form group is created.

The steps below are not to be performed as an in-class exercise. They are for reference only.

Create a compound attribute

1. On the Project Tables View tab, right-click the header of the project table you want to use as the primary lookup table for the attribute. Then select **Create Attribute**.
2. In the MicroStrategy Architect window, in the box, type a name for the attribute.
3. In the Create New Form Expression window, define the ID form expression as desired, then click **OK**.
4. On the Project Tables View tab, select the column you want to add to the attribute you just created.
5. Drag the column to the attribute you just created. This action creates a new description form with a DESC category.

If the column is already used in another attribute:

- a. Right-click the attribute and add a new attribute form.
- b. In the Properties Pane, for the new form, change the **Category** to **ID**.

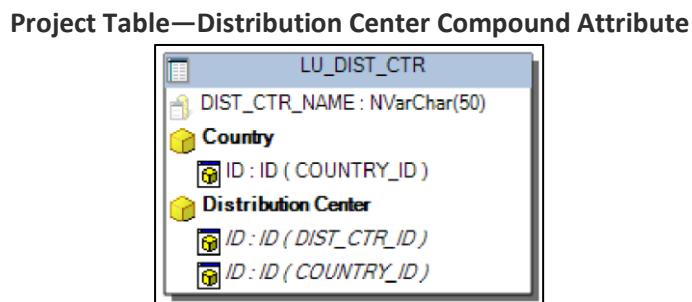
- c. In the message window that prompts you to create a form group, click **Yes**.
- 6. In the Properties pane, modify the **Category** property of the new description form to **ID**.
- 7. In the message window that prompts you to create a form group, click **Yes**.



- 8. In the Properties pane, modify the **Name** property of the new form to **ID**.

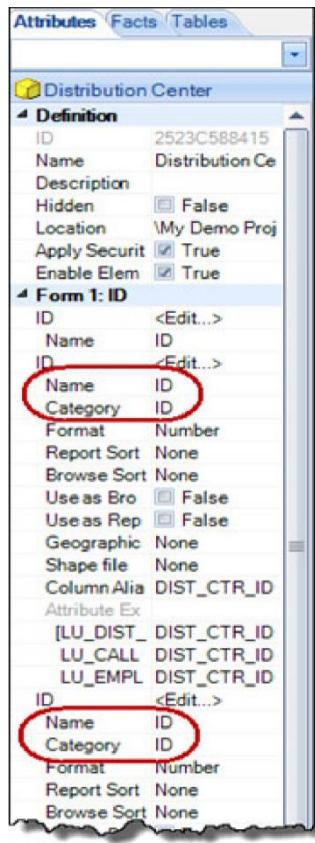
Using ID as the form group name is not a requirement, however, this naming convention is the easiest for users to understand.

The following image shows a table with Distribution Center as a compound attribute:



The following image shows the Properties pane for the Distribution Center attribute with form group:

Properties Pane of a Compound Attribute—Distribution Center



Reusing attribute columns

After you map an attribute to a column in a table, that column is no longer displayed in the table in Architect. This allows you to keep track of the unused columns in your data warehouse and streamline the attribute creation process.

In some cases, however, you may want to reuse columns. For example, consider the following scenario:



In this example, two attributes use the STATE_ID column in their definitions. If you first create the Store State attribute, you use the STATE_ID column in its ID attribute form expression. As a result, this column would no longer appear in the table.

If you want to reuse a column to create an attribute, use the Create New Form Expression window, as outlined in the following steps.

The steps below are not to be performed as an in-class exercise. They are for reference only.

Reuse a column to create multiple attributes

1. On the Project Tables View tab, find the project table that contains the column you want to map to the attribute.
2. Right-click the table and select **Create Attribute**.
3. In the MicroStrategy Architect window, in the box, type a name for the attribute, then click **OK**.
4. In the Create New Form Expression window, define the attribute form expression as desired. Any column can be accessed in a table from this window, even those that have already been used for other attributes.
5. Click **OK**.

Creating attribute forms

An attribute form contains supplementary identification and description information about an attribute. When you create attributes you must also create corresponding attribute forms and form expressions. Every attribute must have at least one attribute form, and each form can have any number of attribute form expressions.

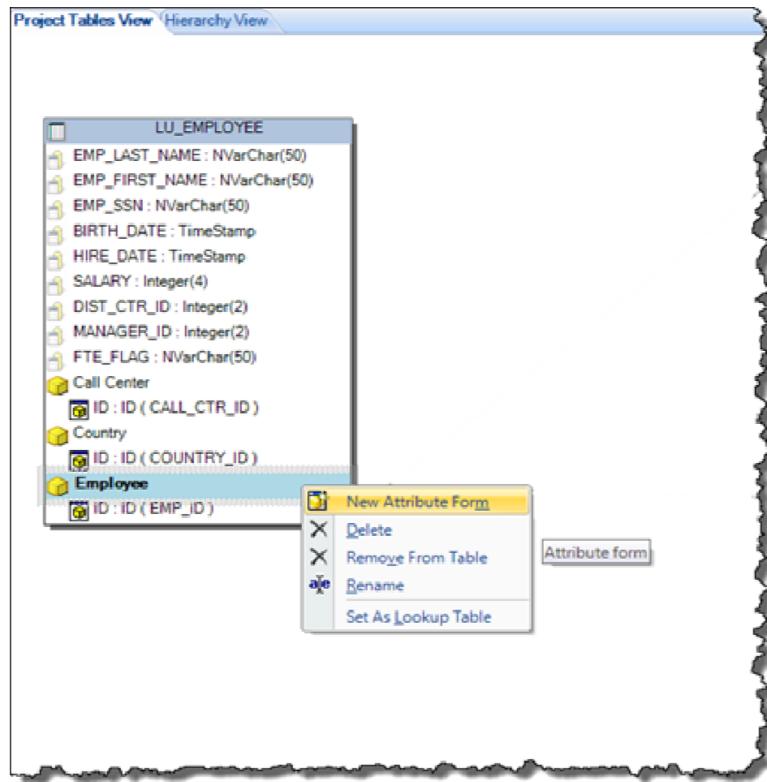
You can create attributes with forms that have simple or derived attribute form expressions. Add as many attribute forms as needed to describe an attribute.

The steps below are not to be performed as an in-class exercise. They are for reference only.

Create a new attribute form

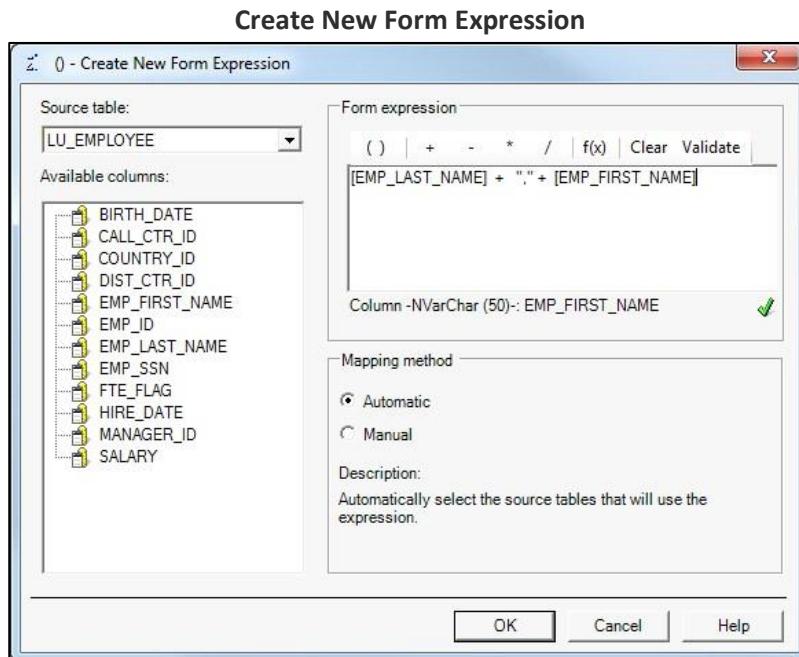
1. On the Project Tables View tab, find the project table that contains the attribute form you want to add.
Attribute forms normally map to an attribute's primary lookup table.
2. In the project table, right-click the desired attribute, and select **New Attribute Form**.

Option for Adding Attribute Forms



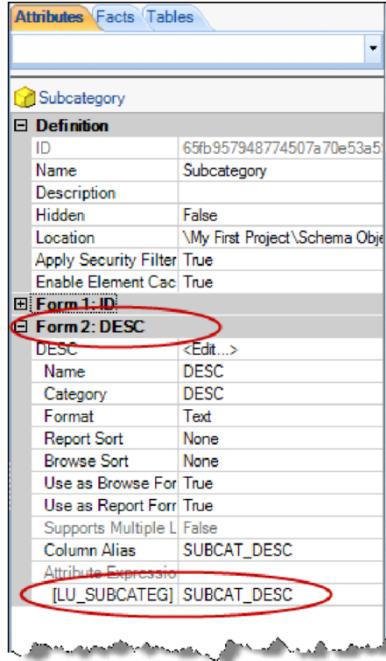
- In the Create New Form Expression window, define a simple or derived form expression and select the mapping method. Then click **OK**.

The following image shows the Create New Form Expression with a simple form expression for the Subcategory attribute:



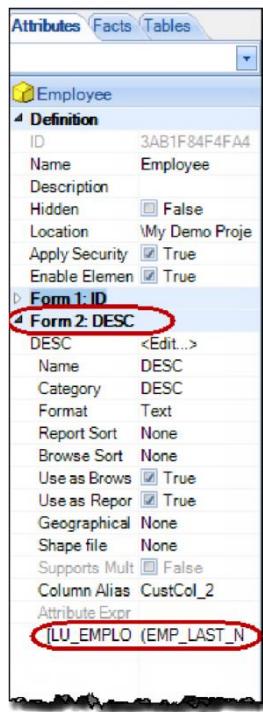
The following image shows the description (DESC) attribute form for the Employee attribute displayed in Architect:

Project Table—DESC Form for Employee Attribute



The following image shows the DESC form for the Subcategory attribute displayed in the Properties pane:

Properties Pane—DESC Form for Subcategory Attribute



If the manual mapping method is selected and you need to map additional tables to the form, the attribute form must be modified.

Modifying attribute forms

After you create an attribute, you can modify it to adapt to changing business requirements. The following aspects of an attribute can be modified:

- Attribute forms
- Attribute form expressions
- Primary lookup table
- Source tables
- Mapping methods
- Column aliases

In Architect, you can modify attributes from the Project Tables View tab or the Properties pane. To access the full range of attribute functions from the Project

Tables View tab, change the display properties to show attribute forms in the project tables.

The steps below are not to be performed as an in-class exercise. They are for reference only.

Modify an attribute form from the Properties pane

1. In the Properties pane, click the **Attributes** tab.
2. In the drop-down list, select the attribute you want to modify.
3. Under the appropriate form category, select the desired form property.

For example, if you are modifying an ID form, the default property name is ID. If modifying a description form, the default property name is DESC.

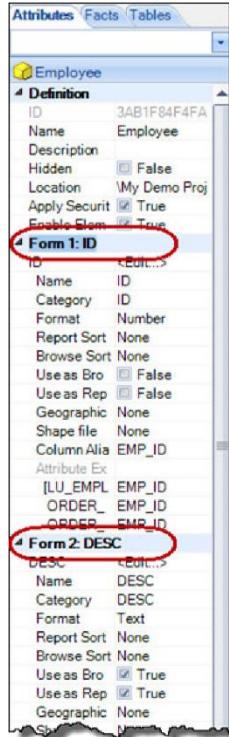
4. Select the expression, then click ... (the Browse button).
5. In the Modify Attribute Form window, update the form as desired.

You can change the source tables used in the attribute form expressions and modify the column alias used in temporary tables. To change the definition of the attribute, you can modify, delete, or create new attribute expressions. The attribute's primary lookup table can also be updated.

6. After completing changes to the attribute form, click **OK**.

The following image shows the attribute form options available in the Properties pane:

Attribute Form Categories



The Properties pane also contains settings that enable you to modify attribute form expressions for specific tables, and update column aliases.

Modifying column aliases

A column alias allows you to specify the temporary table column name and default data type for an attribute form. Attribute forms inherit their data type from the column to which they are mapped. You can modify this data type by changing the column alias.

For example, you may create a derived attribute form based on a column that stores Date data. If you want to store this derived attribute as an Integer, you can modify its column alias.

The steps below are not to be performed as an in-class exercise. They are for reference only.

Modify the column alias for an existing attribute

1. In the Properties pane, click the **Attributes** tab.
2. In the drop-down list, select the attribute you want to modify.
3. Under the desired form category, select the **Column Alias** property.
4. Beside the column alias expression, click ... (the Browse button).
5. In the Column Editor - Column Selection window, modify the existing column alias or create a new one.

Viewing and modifying properties for attributes

Each attribute form has a list of properties that can be modified to reflect changes in your business requirements. In Architect, you can modify these properties in the Properties pane. The following table lists all attribute form property descriptions:

Category	Setting	Description
Definition	ID	GUID that identifies the attribute in the metadata Note: You cannot modify this property.
	Name	Attribute name
	Description	Description of the attribute Note: This property does not contain a value if a description has not been entered.

	Hidden	Determines whether the attribute is a hidden object
	Location	Project folder location for the attribute Note: You cannot modify this property.
	Apply Security Filters	Determines whether security filters are applied when browsing elements for the attribute
	Enable Element Caching	Determines whether elements of the attributes are cached when browsing the attribute
Category	Setting	Description
Form <Number>: <Form Name>	<Form Name>	Provides the Edit option to modify the attribute form
	Name	Name of the attribute form
	Category	Form category for the attribute form
	Format	Format that corresponds to the data type of the attribute form
	Report Sort	Indicates the default sort order for the attribute form when it is displayed on reports
	Browse Sort	Indicates the default sort order for the attribute when browsing the attribute
	Use as Browse Form	Indicates whether the attribute form is displayed when browsing the attribute Note: When you first create an attribute, the ID form defaults to True for this property. If you add a subsequent form (like a DESC form), the value of the new form defaults to True and the value of the ID form automatically changes to False.
	Use as Report Form	Indicates whether the attribute form is displayed when the attribute is shown on reports Note: When you first create an attribute, the ID form defaults to True for this property. If you add a subsequent form (like a DESC form), the value of the new form defaults to True and the value of the ID form automatically changes to False.

	Geographical Role	Indicates the attribute form's geographical data compatibility with various MicroStrategy mapping features
	Shape File	Indicates the shapes used to display the attribute form on various MicroStrategy mapping components
	Column Alias	Column alias for the attribute form
Form <Number>: <Form Name>	Supports Multiple Languages	Indicates if an attribute form is configured to support SQL-based data internationalization Note: This property is not available for ID attribute forms.
	Attribute Form Expressions	Source tables for the attribute form and the columns to which it maps in each table Note: Each table to which an attribute form expression is mapped is listed separately for this property.

An attribute contains a distinct set of properties for each of its forms. In Architect, the form names are listed in the following format: Form 1: <Form Name>, Form 2: <Form Name>, and so on.

View or modify the properties of an attribute

1. In Architect, do one of the following:
 - On the Project Tables View tab, find a project table that contains the attribute you want to view or modify. In the project table, click the attribute. The attribute is displayed in the Properties pane.
 - In the Properties pane, click the **Attributes** tab. From the drop-down list, select the attribute you want to view or modify.
2. In the Properties pane, view or modify the attribute properties using one of the following methods:
 - Modify the property by entering the desired values into text boxes or by selecting values from drop-down lists.
 - Click the property or its current value to display the Browse button (...).
Click **Browse** to display the appropriate property editor window.

Displaying attribute forms

As you learned earlier, attribute forms enable report analysts to display various information about an attribute. To customize this display behavior, you can define the forms that display by default when the attribute is browsed or added to a report.

Report display forms are displayed when an attribute is present on a report.

Browse forms display when you browse an attribute in a hierarchy in the Data Explorer. By default, attribute ID forms are added to the report display. IDs are added to the browse display only if description forms have not been created.

Description forms are added to the default report display and browse display by default. When you create a new form, you can omit it from the report or browse display lists, as desired.

Analysts can also specify the forms that an attribute displays on a particular report. To streamline display settings, set the most commonly displayed forms at the attribute level. Analysts can then modify how attributes display on various reports, as desired.

Define report display and browse forms for an attribute

1. In Architect, do one of the following:
 - On the Project Tables View tab, find a project table that contains the attribute you want to modify. In the project table, select the attribute. The attribute is displayed in the Properties pane.
 - In the Properties pane, click the **Attributes** tab. From the drop-down list, select the attribute you want to modify.
2. To display the form when browsing the attribute, select the **Use as Browse Form** check box.
3. To display the form on reports, select the **Use as Report Form** check box.

By default, the ID form is displayed on reports. If an additional form is created, the value of the new form defaults to True, and the value of the ID form automatically changes to False.
4. Repeat these steps to modify the report display and browse display for the remaining forms.

The following image shows the Properties pane with attribute form display options highlighted:

Properties Pane—Attribute Form Display

Employee

Definition

- ID: 3AB1F84F4FA
- Name: Employee
- Description:
- Hidden: False
- Location: \My Demo Proj
- Apply Securit: True
- Enable Elem: True

Form 1: ID

- ID: <Edit...>
- Name: ID
- Category: ID
- Format: Number
- Report Sort: None
- Browse Sort: None
- Use as Bro: False
- Use as Rep: False
- Geographic: None
- Shape file: None
- Column Alias: EMP_ID
- Attribute Ex:
 - [LU_EMPL] EMP_ID
 - ORDER_ EMP_ID
 - ORDER_ EMP_ID

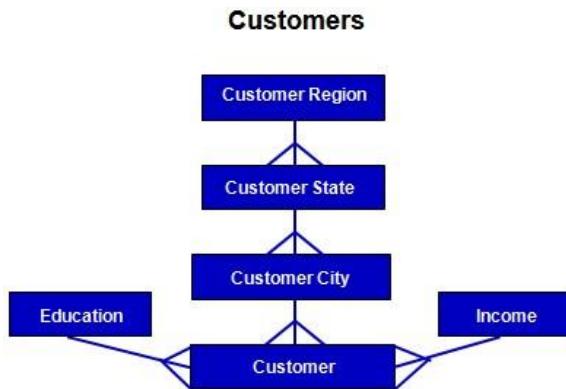
Form 2: DESC

- DESC: <Edit...>
- Name: DESC
- Category: DESC
- Format: Text
- Report Sort: None
- Browse Sort: None
- Use as Bro: True
- Use as Rep: True
- Geographic: None
- Shape file: None
- Supports M: False
- Column Alias: CustCol_2
- Attribute Ex:

Creating attribute relationships

Attribute relationships define the cardinality and structure of attributes in a hierarchy. This information is used to display the order of attributes in a hierarchy when it is browsed, and to define how attributes work together to calculate the level of metrics in a report. When you create the attributes in a hierarchy, the final step is to specify the relationships between attributes, as you previously defined in the logical data model. For example, the relationships between attributes in the Customer hierarchy is defined by the following diagram:

Customer Hierarchy



This logical data model uses Crow's Foot Notation to denote the cardinality between attributes in the hierarchy. For example, Customer Region has a one-to-many relationship with Customer State.

Creating attribute relationships

To create attributes in Architect, use one of the following methods.

- **Manual attribute relationship** creation requires you to identify how attributes are related and specify cardinality based on your data warehouse model. This method gives you maximum control over attribute relationships, but is more labor intensive.
- **Automatic attribute relationship** creation allows Architect to automatically identify and create attribute relationships based on a set of rules you define. After the attribute relationships are created, you must review them to ensure they are complete and accurate. This method generally saves time because it minimizes the amount of manual work required.

The method you choose depends on the physical structure of the tables and columns in your data warehouse, the extent to which users want to analyze various relationships, and your understanding of the data warehouse structure.

Creating attribute relationships manually

If automatic relationship recognition is disabled when you create attributes, the

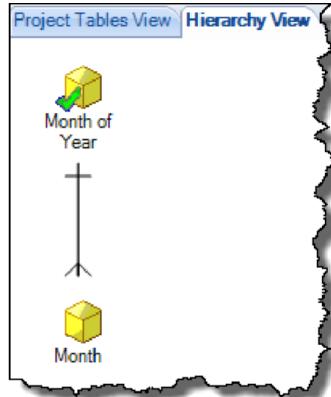
Hierarchy View tab in Architect displays each attribute without any relationship indicators. The following image shows the attributes in the Time hierarchy without defined relationships:



Because relationships have not been specified in this hierarchy, all attributes display as entry points (denoted by the green check marks). Once you define relationships, only the top-level attributes that do not have parents are identified as entry points for the hierarchy.

The following image shows the relationship between the Month of Year and Month attributes; the relationship is represented by the line between the attributes:

Relationship between the Month of Year and Month Attributes



To define relationships in Architect, you can simply click and drag from a parent attribute to its child in the Hierarchy View tab. When you select an attribute, the attributes that are child candidates appear brighter than attributes that cannot be selected as children. Architect considers attributes to be potentially related if their ID forms appear in the same project table.

The steps below are not to be performed as an in-class exercise. They are for reference only.

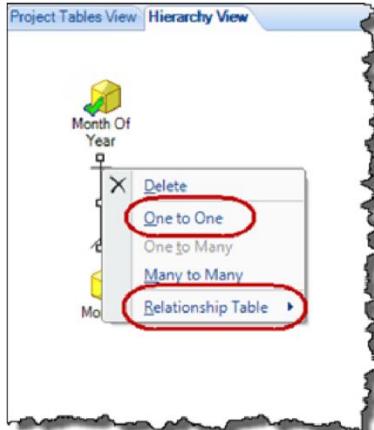
Manually create an attribute relationship

1. On the Hierarchy View tab, click a parent attribute and drag the cursor to the desired child attribute.
When you click and drag the cursor, a line is drawn to link the two attributes.
2. To change the relationship type, right-click the relationship line and select the appropriate relationship type.

One-to-many is the default relationship type.

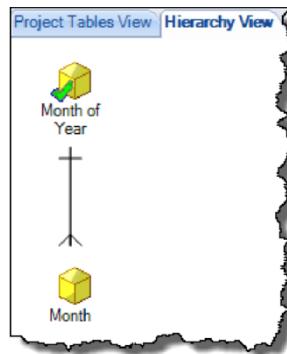
- To change the relationship table, right-click the relationship line, point to **Relationship Table**, and select the desired table.

Options for Selecting Attribute Relationship Types and Relationship Tables



The following image shows the relationship between the Month of Year and Month attributes:

Relationship between the Month of Year and Month Attributes

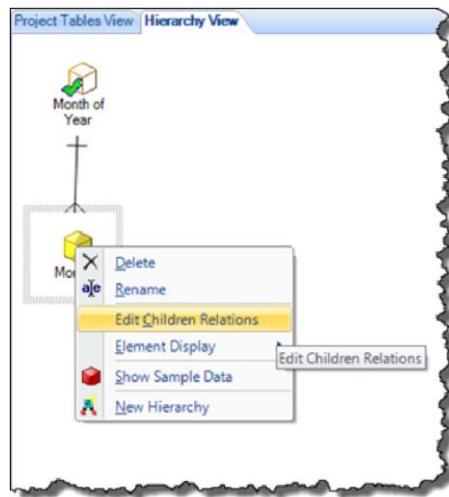


The relationship line between attributes indicates the selected relationship type: one-to-one, one-to-many, or many-to-many. You can hover over the relationship line to display the relationship table name.

You may find that some attribute relationships may need to be modified based on evolving user requirements or creation errors. Once a relationship between attributes is established in Architect, you can modify the relationship at any time.

To do this, right-click the attribute and select **Edit Children Relations**. In the Children Relations window, select the desired relationship for a given child attribute in the Relationship type column. The following image shows the option for modifying parent-child relationships:

Option for Modifying Child Attributes



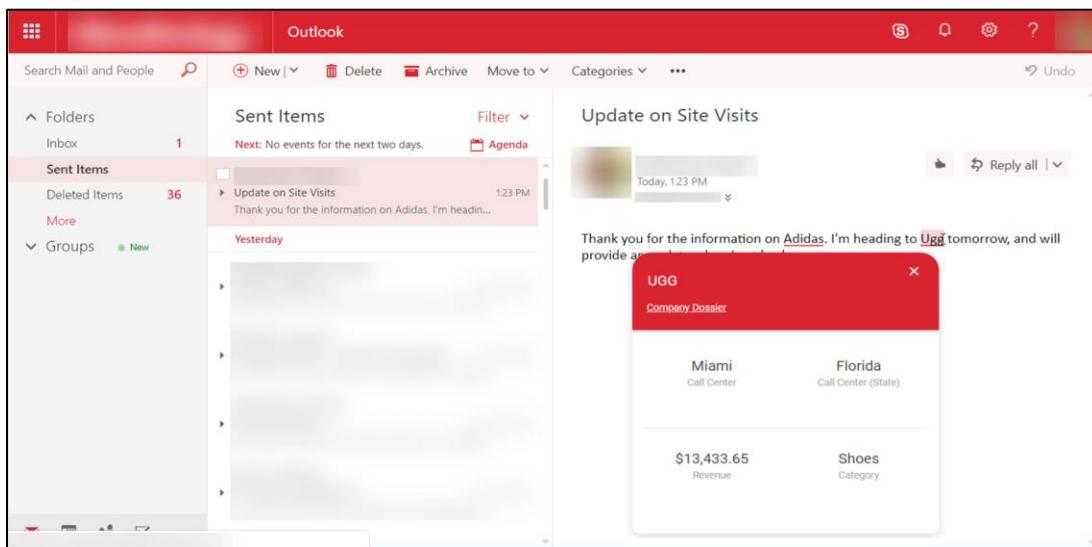
The following image shows the Children Relations window:



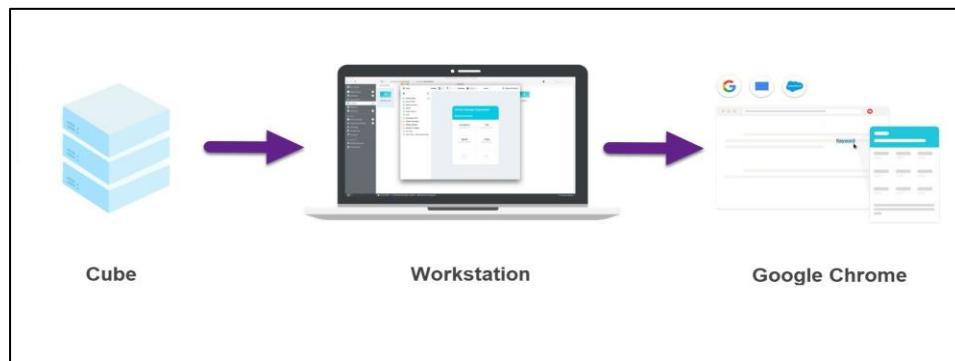
If you find that an attribute relationship is no longer valid, you can also delete a relationship line between two attributes on the Hierarchy View tab. To do this, right-click the relationship line and select **Delete**.

Defining attributes to display on HyperIntelligence Cards

HyperIntelligence displays informative cards in your Chrome browser as you browse the web and interact with web apps that contain matches to your data. For example, you can create a card that displays detailed information about your client organizations. When a user browses the web, or uses a web application like an email app, the card is displayed when she scrolls over a client organization name, as displayed in the following image.



You create and format HyperIntelligence Cards in Workstation and pull data from cubes, which are optimized to deliver data with high performance. The HyperIntelligence Card workflow is displayed in the following image.



The header of the HyperIntelligence Card contains a keyword attribute, which serves as the basis for analysis. For example, if you want to create a card that displays information about your customers, you would use the Customer attribute in the header. You would then display related attributes and metrics in the card to convey the desired information to users.

To ensure that data is displayed correctly on your card, you must ensure that the keyword attribute has one-to-one or one-to-many relationships with any other attributes you plan to display on the card. For example, Customer and Member ID likely have a one-to-one relationship. Customer and Invoice likely have a one-to-many relationship.

Although your card is displayed in Chrome based on the identification of a keyword attribute element, you may want the card to display under additional circumstances. For example, if your keyword attribute is the name of a football team, you may want the card to display when the team's formal name is encountered as well as situations where the team's nickname is identified. You can do this by creating a secondary attribute form for your keyword attribute. You can keep this secondary attribute form hidden or display it on your card.

You can also specify the attribute form to display for each attribute you place on the body of the card. For example, your Location attribute may have Latitude, Longitude, and Name forms. To ensure that card viewers can quickly view and process the location, you would display the Name form.

When you create your dataset, ensure that your attributes have forms that are suitable to display on the card. To ensure your cards are displayed in all possible cases, ensure that your keyword attribute has forms that capture all desired trigger scenarios.

Exercises

Exercise 4.1: Creating attributes and relationships

In this exercise, you will use Architect to manually create attributes in My Demo Project. The following high-level steps and general instructions are provided for the overall work you must complete. Detailed steps for two of these attributes are provided below these general exercise parameters, to help get you started.

- Create the following attributes in their corresponding layers:

Geography Attributes

Source Tables	Column Name	Attribute Name
lu_call_ctr	CALL_CTR_ID	Call Center
lu_region	REGION_ID	Region
lu_country	COUNTRY_ID	Country
lu_dist_ctr	COUNTRY_ID, DIST_CTR_ID	Distribution Center
lu_employee	EMP_ID	Employee

Time Attributes

Source Tables	Column Name	Attribute Name
lu_day	DAY_DATE	Day
lu_month	MONTH_ID	Month
lu_month_of_year	MONTH_OF_YEAR	Month of Year
lu_quarter	QUARTER_ID	Quarter
lu_year	YEAR_ID	Year

Distribution Center is a compound attribute.

- After creating these attributes, create the following description forms for selected attributes, using the following guidelines:

- Use automatic mapping for all attribute form expressions
- As the form expression for each attribute is created, use the Properties pane to verify correct expressions and mapping to the appropriate tables
- Save project work after creating each attribute form

Attribute Name	DESC Form Expression
Call Center	CENTER_NAME
Country	COUNTRY_NAME
Month	MONTH_DESC
Month of Year	MONTH_OF_YEAR_DESC
Quarter	QUARTER_DESC
Region	REGION_NAME

- After creating these attributes and their respective forms, manually create the following child relationships for each attribute, using the following guidelines:

All relationships should be defined on the Hierarchy View tab in Architect.

All relationships are one to many unless another type of relationship is indicated in the table. One-to-one relationships are indicated as 1:1.

Attribute Name	Child Attribute
Country	Region
Country	Distribution Center
Distribution Center	Call Center (1:1)
Call Center	Employee
Month	Day
Month of Year	Month

Quarter	Month
Region	Call Center
Year	Quarter

- After creating the child relationships for these attributes, save and update the project schema.

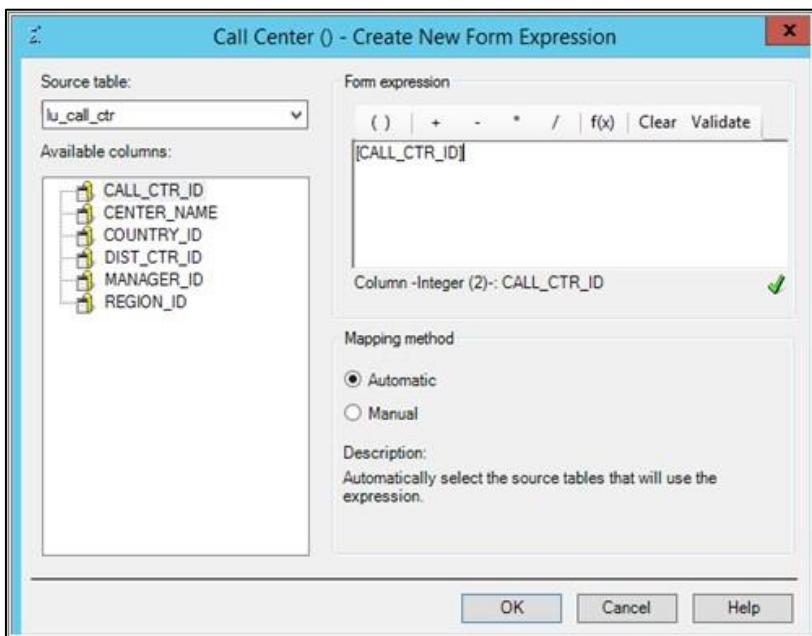
Detailed steps are provided below only for the Call Center attribute and the Distribution Center attribute. Use the same set of steps to create the remaining attributes using the tables and column names provided in the general instructions above.

Create attributes, specify forms, and define relationships

1. In the My Tutorial Project Source, in My Demo Project, from the **Schema** menu, click **Architect**.
If you were logged out, log in to My Tutorial Project Source using the administrator user's credentials.
2. In Architect, click the **Project Tables View** tab.
3. To create the Call Center attribute, on the Home tab, in the Layer area, in the **Layers** drop-down list, select the **Geography** layer.

Create an attribute

4. On the Project Tables View tab, right-click the header of the **lu_call_ctr** table and select **Create Attribute**.
5. In the MicroStrategy Architect window, in the box, type **Call Center**, then click **OK**.
6. In the Create New Form Expression window, under **Available columns**, double-click **CALL_CTR_ID** to move it to the Form expression box.



7. Under Mapping method, ensure **Automatic** is selected, then click **OK**.

A Call Center attribute is created with an ID form that maps to the column CALL_CTR_ID from every project table that contains the column. The new attribute automatically displays in the Properties pane.

8. Repeat the steps in to create the remaining attributes in the Geography and Time layers. (Do not create the Distribution Center attribute at this time; we will create it later.)

Geography Attributes

Source Table	Column Name	Attribute Name
lu_region	REGION_ID	Region
lu_country	COUNTRY_ID	Country
lu_employee	EMP_ID	Employee

Time Attributes

Source Table	Column Name	Attribute Name
lu_day	DAY_DATE	Day
lu_month	MONTH_ID	Month
lu_month_of_year	MONTH_OF_YEAR	Month of Year

lu_quarter	QUARTER_ID	Quarter
lu_year	YEAR_ID	Year

Alternatively, these attributes can be created by right-clicking the appropriate column in the table and selecting **Create Attributes**. Because this method defaults the attribute name to the mapped column name, make sure the attributes are renamed if necessary. To rename an attribute, right-click it and select **Rename**. In the Architect window, in the box, type the attribute name and click **OK**.

Create a description form

1. To create a description form for the Call Center attribute, on the Home tab, in the Layer area, in the **Layers** drop-down list, select the **Geography** layer.
2. On the Project Tables View tab, in the **lu_call_ctr** table, right-click the new **Call Center** attribute and select **New Attribute Form**.
3. In the Create New Form Expression window, in the **Available columns** list, double-click **CENTER_NAME** to define it as a form expression.
4. Under **Mapping method**, ensure **Automatic** is selected.
5. Click **OK**.
6. To display attribute forms in the table, right-click the **lu_call_ctr** table, point to **Properties**, point to **Logical View**, and if it is not already selected, click **Display Attribute Forms**.
7. To create description forms for the remaining attributes, repeat the steps in for the following attributes in their respective lookup tables:

Attribute Name	DESC Form Expression
Country	COUNTRY_NAME
Month	MONTH_DESC
Month of Year	MONTH_OF_YEAR_NAME
Quarter	QUARTER_DESC
Region	REGION_NAME

Alternatively, these attribute forms can be created by dragging the appropriate column to the attribute in the lookup table.

Create a parent-child relationship

1. To create a parent-child relationship for the Call Center attribute, click the **Hierarchy View** tab.
2. Select the **Call Center** attribute and drag the cursor toward the **Employee** attribute. A line with an arrow indicates a relationship is created.
3. Right-click the **Call Center** attribute and select **Edit Children Relations**.
4. In the Children Relations window, in the **Relationship type** drop-down list, make sure the **One to Many** relationship type is selected.
5. Click **OK**.
6. On the Home tab, in the Auto Arrange Hierarchy Layout area, click **Regular** to rearrange the attributes.
7. To create parent-child relationships for the remaining attributes, repeat the steps in for the following attributes:

Attribute Name	Child Attribute
Country	Region
Month	Day
Month of Year	Month
Quarter	Month
Region	Call Center
Year	Quarter

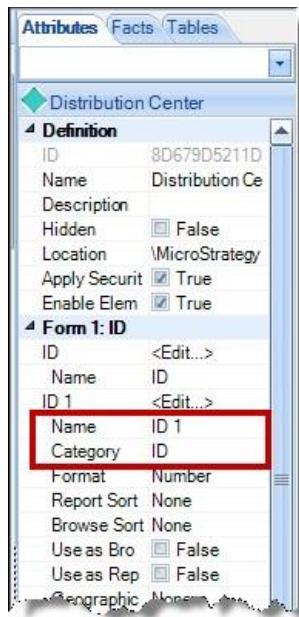
Now that you have created the majority of the attributes for this project, you will create the Distribution Center compound attribute. The steps for this attribute are slightly different because it contains two ID forms.

To create the Distribution Center compound attribute

1. To view the available project tables, click the **Project Tables View** tab.
2. In the Layer area, in the drop-down list, select the **Geography** layer, if not already selected.
3. Right-click the header of the **lu-dist-ctr** table and select **Create Attribute**.
4. In the Architect window, in the box, type **Distribution Center**, and click **OK**.
5. In the Create New Form Expression window, in the **Available columns** list, double-click **DIST_CTR_ID** to define it as a form expression.
6. Under **Mapping method**, ensure **Automatic** is selected, then click **OK** to close the Create New Form Expression window.

Create the attribute forms

7. To create the Country ID form, do the following:
 - a. Right-click the header of the **Distribution Center** attribute and select **New Attribute Form**.
 - b. In the Create New Form Expression window, in the **Available columns** list, double-click **COUNTRY_ID** to add it to the form expression.
 - c. Under **Mapping Method**, ensure **Automatic** is selected. Then click **OK** to close the Create New Form Expression window.
 - d. In the Properties pane, click the Attributes tab. Under Form 2: ID, in the **Name** field, enter **ID**.
 - e. In the Properties pane, click the Attributes tab. Under Form 1: ID, from the **Category** drop down list, select **ID**.

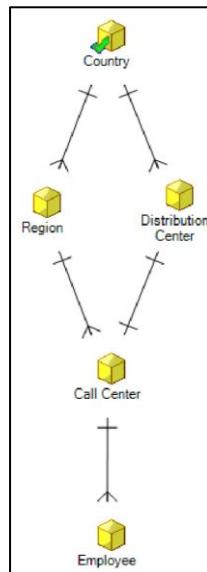


- f. In the Architect window that prompts you to create a form group, click **Yes**.
8. To create the Distribution Center Name form, do the following:
 - a. Right-click the **Distribution Center** attribute and select **New Attribute Form**.
 - b. In the Create New Form Expression window, in the **Available columns** list, double-click **DIST_CTR_NAME** to add it to the form expression.
 - c. Under **Mapping method**, ensure **Automatic** is selected, then click **OK**.
 - d. In the Properties pane, click the Attributes tab. Under Form 2: Name, in the **Name** field, enter **DESC**.
 - e. In the Properties pane, on the Attributes tab, from the **Category** drop-down list, select **DESC**.
9. Define the attribute relationships
 9. To create a parent-child relationship for the Distribution Center attribute, on the Hierarchy View tab, select the **Distribution Center** attribute and drag the cursor towards the **Call Center** attribute.
 10. Right-click the **Distribution Center** attribute and select **Edit Children Relations**.
 11. In the Children Relations window, in the **Relationship type** drop-down list, select the **One to One** relationship type. Then click **OK**.
 12. Select the **Country** attribute and drag the cursor towards the **Distribution Center** attribute.
 13. Right-click the **Country** attribute and select **Edit Children Relations**.

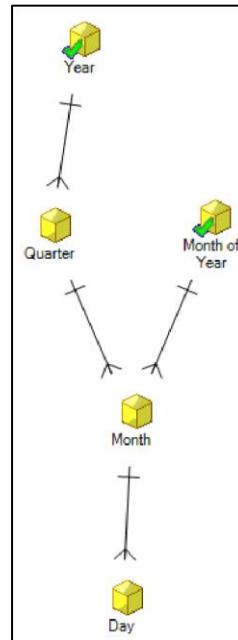
14. In the Children Relations window, in the Distribution Center row's
15. **Relationship type** drop-down list, select the **One to Many** relationship type, if not selected already.

16. To verify the attribute relationships, on the Home tab, in the Auto Arrange Hierarchy Layout area, click **Regular**.

The Geography system hierarchy should look like the following:



The Time system hierarchy should look like the following:



17. Click **Save and Close**. When prompted, update the schema.

CHAPTER 5: WORKING WITH HIERARCHIES

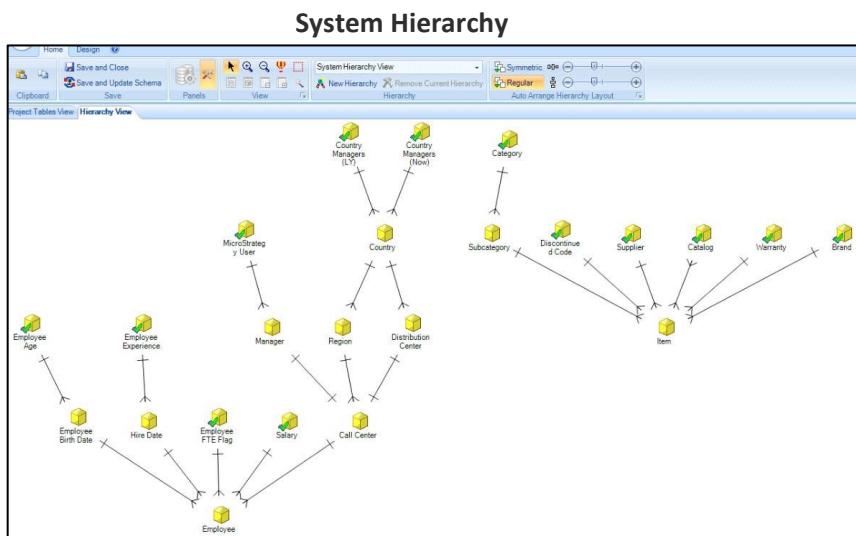
Hierarchies are groupings of related attributes. There are two types of hierarchies in MicroStrategy:

- The system hierarchy includes all attributes in the project, and is automatically created for you.
- User hierarchies enable users to intuitively browse related attributes, and must be manually created.

All attributes in a project: System hierarchy

The system hierarchy includes all the attributes in a project and their respective relationships. It derives its structure from the direct relationships you define between attributes. As you develop attributes and their corresponding relationships, a system hierarchy is automatically created for you.

You can view the system hierarchy in the Hierarchy view tab in Architect:



Attributes that do not have parents serve as entry points in the system hierarchy. Analysts can browse the system hierarchy from the entry points and view directly related attributes. The system hierarchy is automatically updated when attributes or attribute relationships are added, modified, or removed.

Although the system hierarchy is useful for viewing all project attributes, its structure is not conducive to browsing attribute data. The system hierarchy serves as the default drill path used in reports to drill up and down to directly related attributes, but a user hierarchy provides more control over drilling behavior.

Browsing objects intuitively: User hierarchies

A user hierarchy allows you to group attributes based on business requirements, regardless of how data is organized in the logical data model. Create user hierarchies to control how analysts browse attributes in the Data Explorer and drill up or down in reports. For example, you can create a Geography that includes Country, State, and City. This hierarchy enables analysts to re-aggregate report metrics by drilling down from Country to State, drilling up from City to State, and so on.

When you create user hierarchies for browsing, they are visible to users throughout the project. For example, users can access them in the Data Explorer browser, the Object Browser within object editors, and prompts. The following image shows how user hierarchies appear to analysts:

Browsing User Hierarchies

The screenshot displays the MicroStrategy BI Developer Kit interface. On the left, the 'Folder List' shows the 'MicroStrategy Tutorial' section, which contains 'History (21)', 'My Personal Objects', 'Project Builder', 'Project Objects', 'Public Objects', 'Schema Objects', and 'Data Explorer - MicroStrategy Tutorial'. Under 'Data Explorer - MicroStrategy Tutorial', there are several objects including 'Customer', 'Geography', 'Products', and 'Time'. A 'Hierarchies' folder is also present. On the right, the 'Object Browser' window is open, showing a tree structure with 'Customers', 'Geography', 'Products', 'Time', and 'System Hierarchy' under 'Hierarchy'. Below the tree, a table lists 'Name / Type' for each object. The 'Prompts' window is also visible, showing a 'Time' prompt with a dropdown menu and some descriptive text.

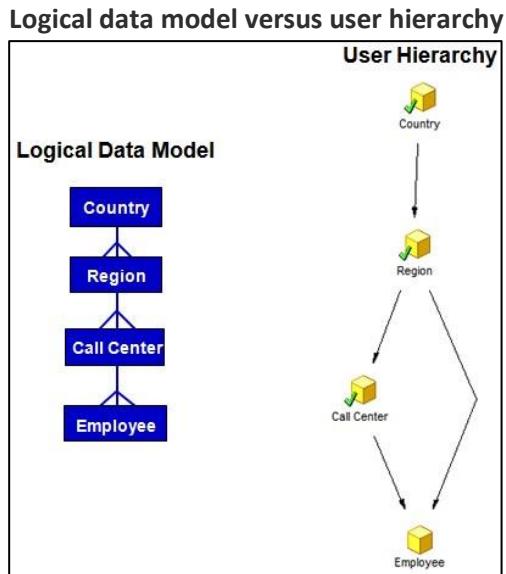
In addition to providing users with a convenient method of browsing attribute data, user hierarchies can also be configured as drill paths for analyzing report data. The following image shows how users can drill to related detail based on user hierarchies within reports:

Drilling on User Hierarchies

The screenshot shows a report table with columns 'Subcategory', 'Metrics', 'Revenue', and 'Units Sold'. The 'Metrics' column has a header 'Revenue' and 'Units Sold'. The 'Subcategory' column lists various categories like 'Art & Architecture', 'Business', 'Literature', etc. A context menu is open over the 'Revenue' header, with 'Drill' selected. The 'Drill' submenu includes 'Sort', 'Attribute Forms', 'Move', 'Remove from Grid', 'Formatting', 'Lock Object', 'Edit...', and 'Remove from Report'. The 'Edit...' option is highlighted with a yellow background. The 'Drill' submenu also has 'Other directions' and 'Down' options, which further expand to show 'Time', 'Products', 'Customers', and 'Geography'.

Design your user hierarchies to accommodate user requirements. For example, if analysts typically browse geography data at the employee level, you can create a user hierarchy that enables users to navigate from the Region attribute, a higher-level attribute in the hierarchy, directly to the Employee attribute, the lowest-level

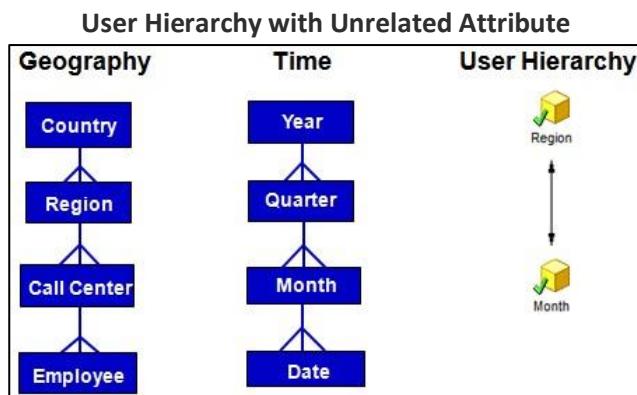
attribute in the hierarchy. The following image shows how the logical data model and the user hierarchy might differ in structure:



The logical data model does not have a direct path between the Region and Employee attributes because Call Center is between them. Because user hierarchies do not need to follow the logical data model, you can accommodate the desired browsing behavior by creating a user hierarchy.

The attributes you add to a user hierarchies do not need to be directly related. Attributes from different hierarchies in the logical data model can be added to a user hierarchy to help you achieve unique browsing behavior.

For example, if users often browse data by Region and Month, you might create a user hierarchy that includes these attributes, even if they are not directly related. The following image shows how a user hierarchy that combines attributes from different hierarchies in the logical data model:



This example demonstrates how attributes from different hierarchies in the logical data model can be combined in a user hierarchy to enable analysts to easily browse and drill between unrelated attributes.

Although user hierarchies are not required in a MicroStrategy project, they help users seamlessly interact with data. To help you develop user hierarchies for your project, consider the unique ways in which your users navigate data, and determine whether those interactions are supported by the system hierarchy.

Creating user hierarchies

In Architect, create user hierarchies from the Hierarchy View tab. The following high-level steps outline the user hierarchy development process:

1. Create the user hierarchy object.
2. Add attributes to the user hierarchy.
3. Define the user hierarchy by specifying the following:
 - Browse attributes
 - Entry points
 - Element display
 - Availability for drilling
 - Attribute filters
4. Customize the sort order for browsing and drilling based on user hierarchies.
5. After you create and define user hierarchies, move them to the Schema Objects\Hierarchies\Data Explorer folder to make them available for browsing.

The following sections describe how each high-level step is performed in Architect.

Creating a user hierarchy object

When you click the Hierarchy View tab in Architect, the toolbar displays the hierarchies' drop-down list. By default, this list only contains the System Hierarchy View. When you create a new user hierarchy it appears in the list. To do this, click **New Hierarchy**.

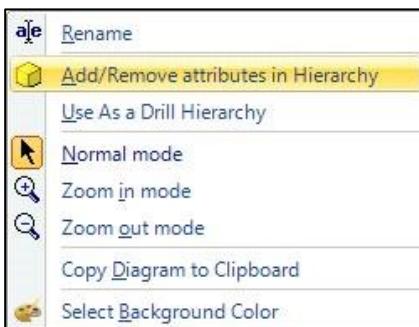
The following image shows the Geography user hierarchy displayed on the Hierarchy View tab:



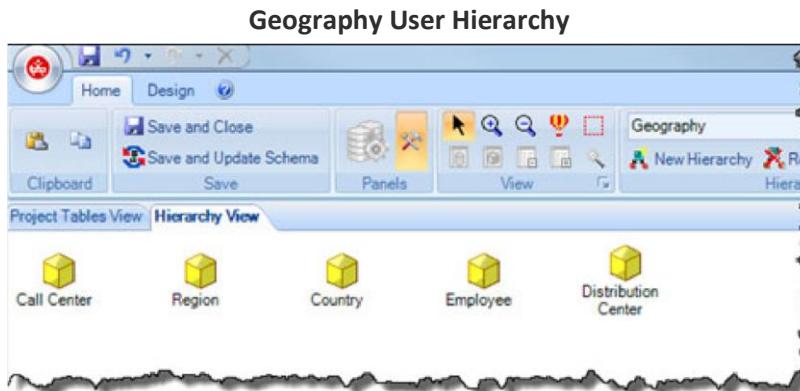
Adding attributes to user hierarchies

The newly created user hierarchy is empty. The next step is to add the attributes you want to include in the hierarchy. To do this, right-click in the empty Hierarchy View tab and select **Add/Remove attributes in Hierarchy**, as in the following image:

Option for Adding Attributes to a User Hierarchy



The following image shows the Geography user hierarchy with several attributes added to it:



Defining user hierarchy settings

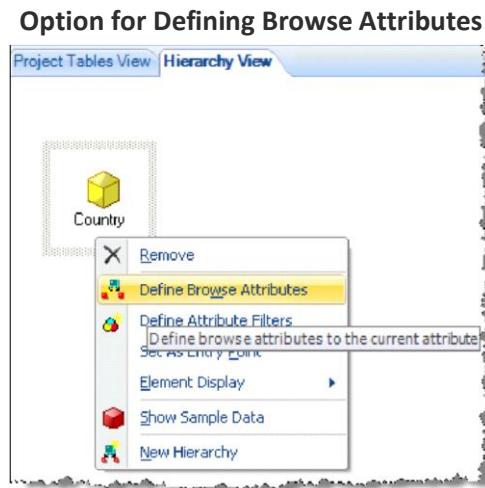
After you add attributes to your user hierarchy, the next step is to define its structure and functionality. To do this, define the browse attributes and entry points. You can also define the element display for attributes, define filters on attributes, and configure user hierarchies for drilling.

Defining browse attributes

Each attribute in a user hierarchy has a corresponding list of connected browse attributes. This list defines the attributes that analysts can browse to from the associated parent attribute in the Data Explorer.

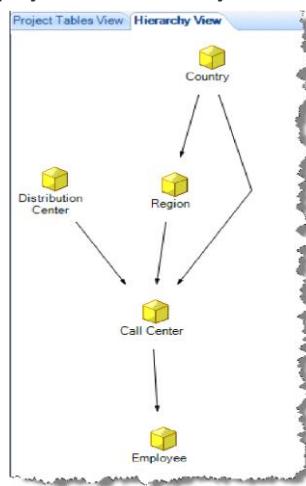
By default, when a user hierarchy is created, only the immediate children of a parent attribute are defined as browse attributes. You can modify the browse attributes to satisfy user requirements.

The following image shows the option for defining browse attributes:



Browse attributes are indicated by a line that connects the two attributes. The following image shows the Geography User Hierarchy with the various browse paths defined:

Geography User Hierarchy—Browse Paths

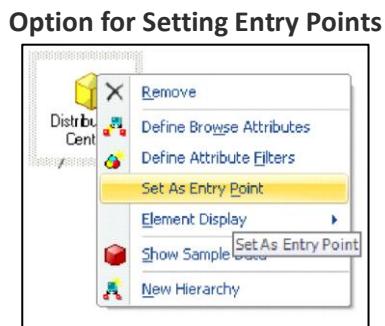


Setting entry points

An entry point serves as a shortcut to an attribute in the Data Explorer. Analysts can begin browsing a user hierarchy from the entry points that you specify. For example, a Geography hierarchy may include Country, Region, State, City, County, and Zip Code. If Zip Code is a commonly accessed attribute, you may choose to set it as an entry point so that analysts do not have to click through the entire hierarchy to reach it.

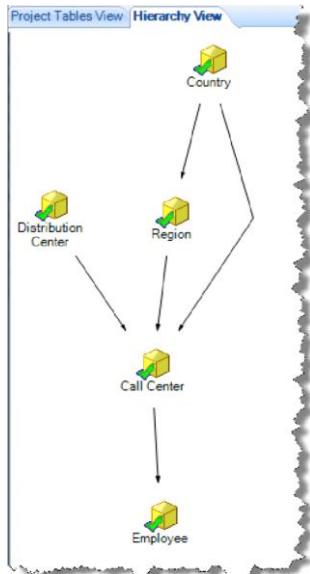
To provide direct access to specific attributes in a user hierarchy, set those attributes as entry points. In the Hierarchy View in Architect, an entry point is indicated by a green check mark on the attribute icon.

The following image shows the option for setting entry points:



The following image shows the Geography user hierarchy with all attributes set as entry points:

Geography User Hierarchy—Entry Points



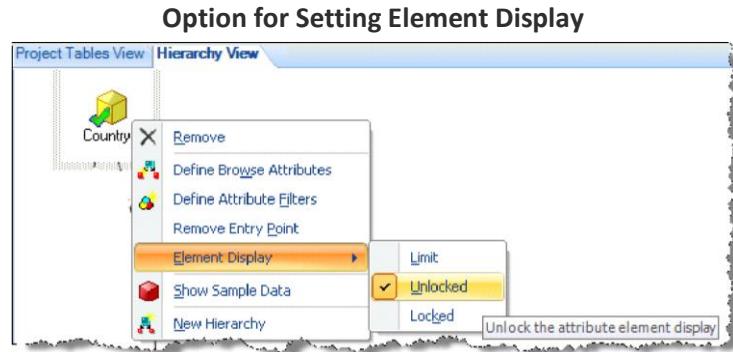
Setting element display

The element display setting determines the extent to which users can browse attribute elements. You have the following options for configuring an attribute's element display:

- **Unlocked:** Display all the elements of an attribute at one time for browsing.
- **Locked:** Cannot display any of the elements of an attribute for browsing.
- **Limit:** Display a specified number of attribute elements for browsing. Users can then either retrieve the next set of elements or return to browsing the previous set of elements.

When you lock or limit an attribute's element display, browsing is only impacted for the specified user hierarchy. The element display setting for an attribute must be specified in each individual user hierarchy.

The following image shows the option for setting the element display:

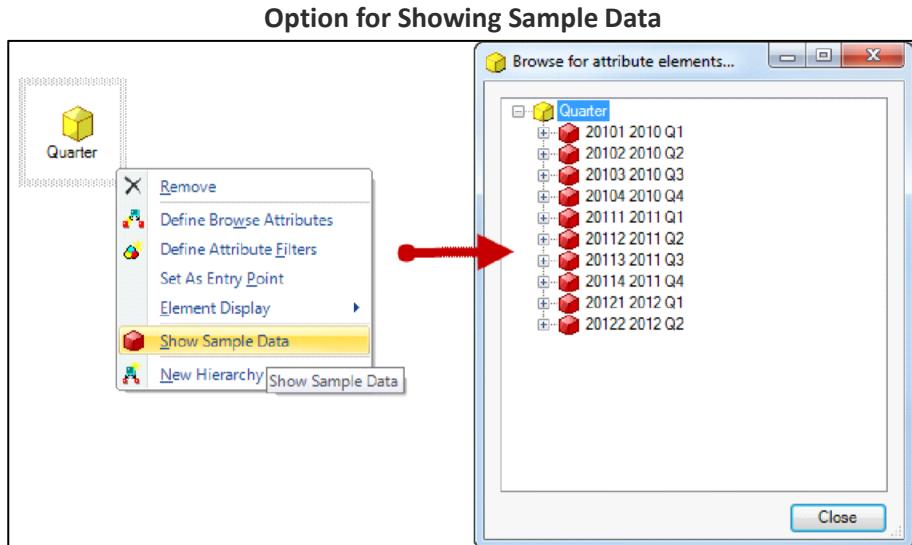


Locked attributes are indicated by a padlock image on the attribute icon.

Showing sample data

Sample data provides a glimpse of the attribute elements stored in the data warehouse. This information can help you identify appropriate display settings for each attribute.

The following image displays the Show Sample Data option and sample elements for the Quarter attribute:



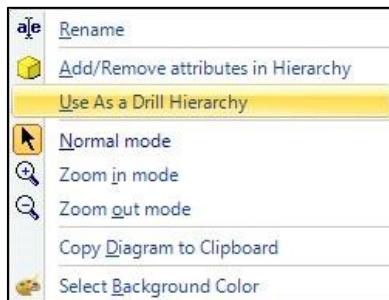
Configuring a user hierarchy for drilling

Drilling allows analysts to aggregate reports to various levels of detail. This enables users to interact with reports to find the specific information they need.

When a user hierarchy is made available for drilling, it displays as a drill path for attributes in reports. When you configure a user hierarchy for drilling, it is displayed on the Other Directions drill menu in reports.

The following image shows the option for configuring a user hierarchy for drilling:

Option for Configuring a User Hierarchy for Drilling



Defining attribute filters

Attribute filters on a user hierarchy control the data that is displayed for a given attribute. A filter on an attribute in a hierarchy works just like a filter on a report. Only attribute elements that match the filter conditions are displayed when users browse the attribute. You can create various types of attribute filters that contain metric qualifications, multiple conditions, or reports in their definition.

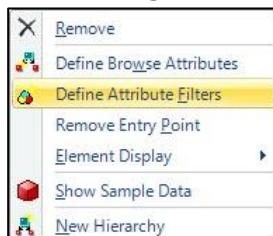
For example, if you browse the Year attribute with a filter for 2010, only the data for 2010 is displayed. If users perform most of their analysis on 2010 data, this filter enables them to quickly access the necessary time frame without sifting through data for other years in the data warehouse.

Attribute filters on a user hierarchy are not a security measure to prevent users from viewing attribute data. Rather, you use them to restrict the results of a browse request to the specific information users need to analyze.

By default, a user hierarchy does not have any filters applied to it. Because filters are defined on a user hierarchy at the attribute level, a hierarchy with multiple entry points requires you to define the same filter on each one to ensure that the filter conditions are applied uniformly on the hierarchy.

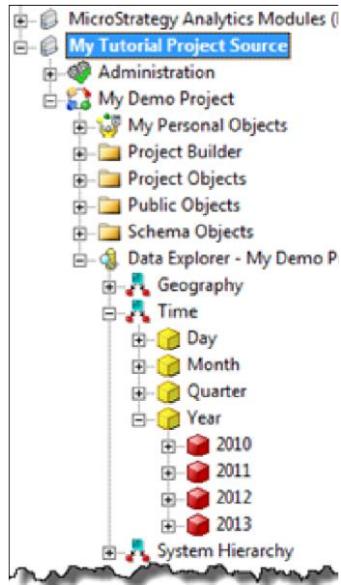
The following image shows the option for defining attribute filters:

Option for Defining Attribute Filters



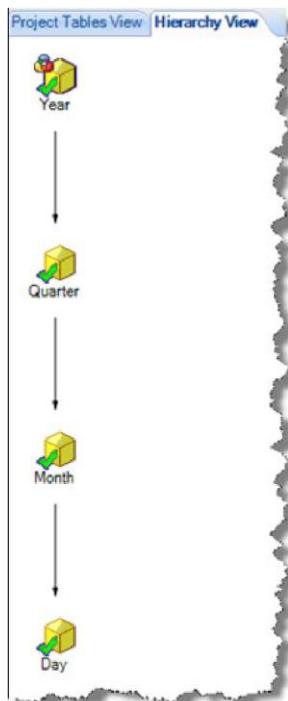
The following image shows the elements of the Year attribute when you browse without a filter:

Browsing Year Attribute without Filter



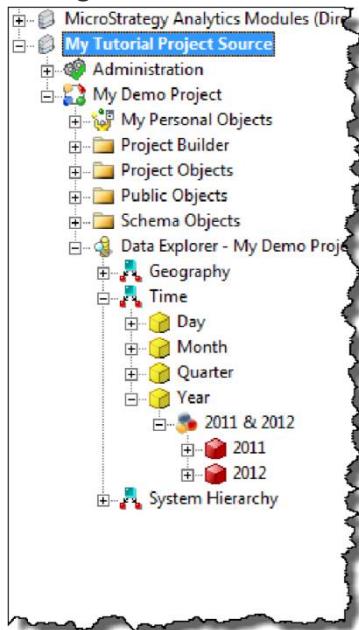
A filter image is displayed on filtered attributes. The following image shows the Time user hierarchy with the filter defined for the Year attribute:

Year Attribute with Filter



The following image shows the elements of the Year attribute with a filter defined for years 2011 and 2012:

Browsing Year Attribute with Filter



Exercises

Exercise 5.1: Creating hierarchies

User hierarchies are groupings of related attributes that enable you to satisfy user browsing and drilling requirements. In this exercise, you will use Architect to create user hierarchies in My Demo Project. Create the following user hierarchies:

- Geography
- Time

The following exercise sections outline the steps required to create each user hierarchy.

Overview: Geography user hierarchy

In the first part of this exercise, you will create a Geography user hierarchy for the geography-related attributes in My Demo Project. The following table includes the attributes and corresponding options you will add to the Geography user hierarchy:

Geography User Hierarchy			
Attribute	Browse Attribute	Entry Point	Element Display
Call Center	Employee	Yes	Unlocked
Country	• Region • Call Center	Yes	Unlocked
Distribution Center	Call Center	Yes	Unlocked
Employee	None	Yes	Unlocked
Region	Call Center Employee	Yes	Unlocked

Create the geography user hierarchy

1. To create the Geography user hierarchy, in the My Tutorial Project Source, in My Demo Project, on the **Schema** menu, click **Architect**.
2. *If you were logged out, log in to My Tutorial Project Source using the administrator user's credentials.*
3. In Architect, click the **Hierarchy View** tab, if not already selected.
4. On the Home tab, in the Hierarchy area, click **New Hierarchy**.
5. In the Architect window, in the box, type **Geography** as the user hierarchy name.
6. Click **OK**. The empty Geography user hierarchy is displayed in the Hierarchy View tab.
7. To add attributes to the Geography user hierarchy, on the Hierarchy View tab, right-click in the empty area and select **Add/Remove attributes in Hierarchy**.
8. In the Select Objects window, in the **Available objects** list, hold CTRL and select the following attributes:
 - Call Center
 - Country

- Distribution Center
- Employee
- Region

9. Click the > button to add the attributes to the **Selected objects** list.

10. Click **OK**.

Define the browse attributes

11. To define the browse attributes for the Geography user hierarchy, on the Hierarchy View tab, in the Geography user hierarchy, right-click the **Call Center** attribute and select **Define Browse Attributes**.

12. In the Select Objects window, in the **Available objects** list, select **Employee**.

13. Click the > button to add the **Employee** attribute to the **Selected objects** list.

14. Click **OK**.

15. Repeat the steps for the following attributes:

- a. Country: Add Region and Call Center browse attributes
- b. Distribution Center: Add the Call Center browse attribute
- c. Region: Add Call Center and Employee browse attributes

16. The Employee attribute does not have any browse attributes.

17. On the Home tab, in the Auto Arrange Hierarchy Layout area, click **Regular** to rearrange the attributes.

18. You may have to switch to the System Hierarchy view and then back to the Geography hierarchy for this setting to take effect. If necessary, click **Regular** again.

Set the entry points for the user hierarchy

19. To set the entry points for the Geography user hierarchy, within the Geography user hierarchy, right-click the **Region** attribute and select **Set As Entry Point**.

20. Repeat the previous step to set the entry points for the following user hierarchies:

- a. Call Center
- b. Country
- c. Distribution Center
- d. Employee

21. Because the element display for all the attributes in the Geography hierarchy is unlocked, you do not need to define element display settings.

Configure the Geography user hierarchy for drilling

22. In the Geography user hierarchy, click anywhere in the empty area to clear any attributes that may have been selected.
23. Right-click an empty space and select **Use As a Drill Hierarchy**. Analysts can now use the Geography hierarchy as a drill path in reports.

Update the project schema

24. On the Home tab, click **Save and Update Schema** and then click **Update** to update the project schema.

Overview: Time user hierarchy

In this part of the exercise, use Architect to create a Time user hierarchy for the time-related attributes in My Demo Project. Configure the Time user hierarchy for drilling as well as browsing.

The following table includes the attributes and corresponding settings you will add to the Time user hierarchy:

Time User Hierarchy			
Attribute	Browse Attribute	Entry Point	Element Display
Day	None	Yes	Unlocked
Month	Day	Yes	Unlocked
Month of Year	Month	Yes	Unlocked
Quarter	Month Day	Yes	Unlocked
year	Quarter Month	Yes	Unlocked

After you save and update the schema, in Developer, move all the user hierarchies you created from the Schema Objects\Hierarchies folder to the Schema Objects\Hierarchies\Data Explorer folder.

Create the Time user hierarchy

The following instructions are written at a general level to help you test your understanding.

1. Create a new Time user hierarchy.
2. To add attributes to the Time user hierarchy, on the Hierarchy View tab, in the Time user hierarchy, right-click an empty area and select **Add/Remove attributes in Hierarchy**.
3. In the Select Objects window, select the following attributes to include in the Time user hierarchy:
 - a. **Day**
 - b. **Month**
 - c. **Month of Year**
 - d. **Quarter**
 - e. **Year**
4. Click **OK**.

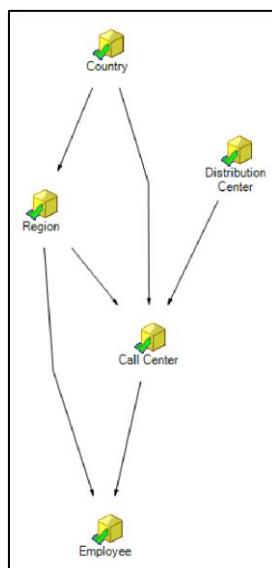
Define the browse attributes

5. To define the browse attributes for the Time user hierarchy, on the Hierarchy View tab, in the Time user hierarchy, right-click the **Month** attribute and select **Define Browse Attributes**.
6. In the Select Objects window, in the **Available objects** list, select **Day**.
7. Click the > button to add the **Day** attribute to the **Selected objects** list.
8. Click **OK**.
9. Repeat the steps for the following attributes:
 - a. Month of Year: Add the Month browse attribute
 - b. Quarter: Add Month and Day browse attributes
 - c. Year: Add Quarter and Month browse attributes The Day attribute does not need any browse attributes.
10. Auto-arrange the attributes within the hierarchy.

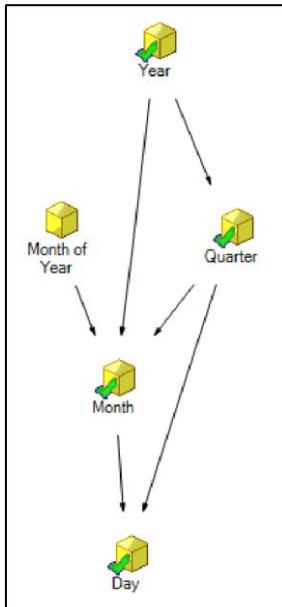
11. On the Hierarchy View tab, in the Time user hierarchy, set the **Day**, **Month**, **Quarter**, and **Year** attributes as entry points.
12. To set the element display, in the Time user hierarchy, right-click the **Day** attribute, point to **Element Display**, and select **Limit**.
13. In the **Limit** box, type **31**.
14. Click **OK**.

Configure the Time user hierarchy for drilling

15. In the Time user hierarchy, click anywhere in the empty area to clear any attributes you may have selected.
16. Right-click an empty space and select **Use As a Drill Hierarchy**. Analysts can now use the Time hierarchy as a drill path in reports.
17. The Geography user hierarchy should resemble the image below:



The Time user hierarchy should resemble the image below:



Update the project schema

18. Click **Save and Close** and then update the project schema and exit Architect.

Save the hierarchies

19. To save the Geography and Time hierarchies in the Data Explorer folder, in Developer, browse to the Schema Objects\Hierarchies folder. Drag the **Geography** and **Time** hierarchies to the Data Explorer folder to make them available for browsing.

The hierarchies do not display in the Schema Objects\Hierarchies folder unless you saved and updated the schema. It may be necessary to refresh the Developer display to view them. To refresh the Developer display, browse to the Hierarchies folder and press **F5**. To view the hierarchies in the Data Explorer for the project, in the Folder List, right-click **Data Explorer - My Demo Project** and select **Refresh**.

CHAPTER 6: MANAGING THE MICROSTRATEGY SCHEMA - FACTS

The MicroStrategy schema development workflow is an iterative process. After you finish building the initial version of the schema, you will need to monitor users, understand how they are using the platform, track business changes, identify data warehouse updates, and adapt your schema accordingly.

As your business evolves over time, the information that you store in your data warehouse and your user requirements will also change. To help you adapt to these changes, you will learn to manage the MicroStrategy schema.

To manage the MicroStrategy schema, you will perform the following tasks:

- Add and modify facts.
- Modify attribute forms.
- Create aggregate tables.

Modifying facts

As your data warehouse evolves, you will need to update the facts in your project to reflect changes to the physical data structure. To modify facts in Architect, update the following components:

- Fact expressions
- Source tables
- Mapping methods
- Column alias

In Architect, you can modify facts in either the Project Tables View tab or the Properties pane. The Project Tables View tab provides access to all parts of a fact. The Properties pane provides quick access to only the column alias and individual fact expressions.

Use the Project Tables View tab to modify facts when you need to perform any of the following tasks:

- Modify multiple expressions for a single fact
- Modify a single fact expression for all the tables to which it maps
- Create new fact expressions
- Delete existing fact expressions
- Modify source tables
- Modify column aliases

Modifying fact column aliases

The column alias specifies the data type that the MicroStrategy Engine uses for a fact when it generates SQL for temporary tables. Every fact has a default column alias, regardless of the type or number of expressions defined for it.

If necessary, you can modify the column alias for an existing fact. For example, you can change the column alias name to something that is more meaningful to you. This can help you analyze SQL for complex reports.

By default, a fact inherits its data type from the column that is used to define it. For example, if you map a fact to a QTY_SOLD column that has an Integer data type, Architect automatically creates a corresponding column alias called QTY_SOLD with an Integer data type.

If a fact maps only to a derived expression, Architect creates a custom column alias. Custom column aliases use the naming convention CustCol_<n> where “Cust” stands for custom, “Col” stands for column, and n is a number. The first custom column alias Architect creates in a project is CustCol_1, the next is CustCol_2, and so on.

If you define a fact using multiple expressions, the column alias uses the column name and data type of the first expression you create. If the first expression you create is a derived expression, Architect creates a custom column alias as described above.

You can also modify the default column alias data type. For example, you may create a fact defined as the difference between a start date and expire date, as in the following expression:

[EXPIRE_DATE] - [START_DATE]

The column alias for this fact automatically uses a TimeStamp data type because the EXPIRE_DATE and START_DATE columns use the TimeStamp data type. However, the result of the expression (the difference between the two dates) produces an integer.

The disparity in data types for this fact can be problematic when the MicroStrategy Engine inserts values into a temporary table. The Engine uses a TimeStamp data type to define the fact column in the temporary table and then tries to insert integer numbers into the column. While this may not be a problem for some database platforms, it can cause an error in others. To eliminate the conflicting data types, you can modify the column alias data type from TimeStamp to Integer.

Viewing and modifying properties for facts

In addition to the column alias, each fact has properties such as description, location, expression, and so on. View fact properties in the Properties pane. You can also modify many of these properties. The following table lists these properties and their descriptions:

Category	Property	Description
Definition	ID	GUID that identifies the fact in the metadata Note: You cannot modify this property.
	Name	Name of the fact
	Description	Description of the fact Note: This property does not contain a value if a description has not been entered.
	Hidden	Determines whether the fact is a hidden object
	Column Alias	Column alias for the fact
	Location	Project folder location for the fact Note: You cannot modify this property.
Fact Expressions	<Table Name>	Table column that the fact maps to Note: There is a separate property for each table that the fact is mapped to.

Using the Properties pane is the quickest method to modify a column alias.

The steps below are not to be performed as an in-class exercise. They are for reference only.

Modify a column alias

1. In the Properties pane, click the **Facts** tab.
2. In the drop-down list, select the fact to view or modify.
3. In the Properties pane, click the properties to modify, and modify the fact properties as desired. Some property values can be modified through text boxes or drop-down lists. For other properties, selecting the property or its current value displays the Browse icon (...). You can click the Browse icon to open a window and modify the property.

Exercises

Exercise 6.1: Modify facts in the project

As your business evolves, you may need to add new tables that contain fact data to your data warehouse. To reflect these changes in MicroStrategy, you must update your facts with expressions to retrieve the new data.

In this exercise, use Architect to modify the Cost, Revenue, and Units Sold facts you created in My Demo Project. Add the following expressions to your facts:

Fact	Expression	Source Table
Cost	QTY_SOLD * UNIT_COST	order_detail
	ORDER_COST	order_fact
Revenue	QTY_SOLD * (UNIT_PRICE - DISCOUNT)	order_detail
	ORDER_AMT	order_fact
Units Sold	QTY_SOLD	order_detail
		order_fact

- Use automatic mapping for all fact expressions.
- After these new expressions are created, save and update the project schema.

There are separate sets of instructions for the Cost, Revenue, and Units Sold facts.

Modify the Cost fact

Create the second expression for the Cost fact

1. In the My Demo Project, from the **Schema** menu, click **Architect**.
2. On the Project Tables View tab, in any project table that contains the Cost fact (such as `customer_sls`), right-click **Cost** and select **Edit**.
3. In the Fact Editor, on the Definition tab, click **New**.

4. In the Create New Fact Expression window, in the **Source table** drop-down list, select **order_detail**.
5. In the **Fact Expression** box, create the following expression: **QTY_SOLD * UNIT_COST**.
6. Under **Mapping method**, ensure **Automatic** is selected.
7. Click **OK**.

Create the third expression for the Cost fact

8. In the Fact Editor window, on the Definition tab, click **New**.
 9. In the Create New Fact Expression window, in the **Source table** drop-down list, select **order_fact**.
 10. In the **Fact Expression** box, create the following expression: **ORDER_COST**.
 11. Under **Mapping method**, keep **Automatic** selected.
 12. Click **OK**.
 13. In the Fact Editor window, click **OK**.
 14. **Save the changes**
 15. Click **Save and Update Schema**.
 16. Click **Update** to save the changes that you made to the project. Keep the project open in Architect so you can modify the next fact.
-

Modify the Revenue Fact

Create the second expression for the Revenue fact

1. On the Project Tables View tab, in any project table that contains the Revenue fact (such as **customer_sls**), right-click **Revenue** and select **Edit**.
2. In the Fact Editor window, on the Definition tab, click **New**.
3. In the Create New Fact Expression window, in the **Source table** drop-down list, select **order_detail**.
4. In the **Fact Expression** box, create the following expression: **QTY_SOLD * (UNIT_PRICE - DISCOUNT)**.

5. Under **Mapping method**, keep **Automatic** selected.

6. Click **OK**.

Create the third expression for the Revenue fact

7. In the Fact Editor window, modify the Revenue fact to create a third expression: **ORDER_AMT**.

8. *You can find this column in the ORDER_FACT table.*

9. Use **Automatic** as the mapping method.

10. Click **OK**.

11. In the Fact Editor window, click **OK**.

Save the changes

12. Click **Save and Update Schema**.

13. Click **Update** to save the changes that you made to the project. Keep the project open in Architect so you can modify the next fact.

Modify the Units Sold fact

Create the second expression for the Units Sold fact

1. On the Project Tables View tab, in any project table that contains the Units Sold fact, right-click **Units Sold** and select **Edit**.

2. In the Fact Editor window, on the Definition tab, click **New**.

3. In the Create New Fact Expression window, in the **Source table** drop-down list, select **order_detail**.

4. In the **Fact Expression** box, create the following expression: **QTY_SOLD**.

5. Under **Mapping method**, ensure **Automatic** is selected, and click **OK**.

6. In the Fact Editor window, click **OK**.

Save the changes

7. On the Home tab, click **Save and Update Schema**.

8. In the Update Schema window, ensure the following check boxes are selected:
 - a. Update schema logical information
 - b. Recalculate table keys and fact entry levels
 - c. Recalculate table logical sizes
9. Click **Update**.

Exercise 6.2: Add a new fact to the project

As your MicroStrategy project evolves, users may request new facts to be created.

In this exercise, use Architect to create the Profit fact in My Demo Project. Create the following expressions for this fact:

Fact	Expression	Source Table
Profit	TOT_DOLLAR_SALES - TOT_COST	city_subcateg_sls customer_sls
	QTY SOLD * (UNIT_PRICE - DISCOUNT - UNIT_COST)	order_detail
	ORDER_AMT - ORDER_COST	order_fact

- Use automatic mapping for all fact expressions.
- After this fact is created, save and update the project schema.

Create the Profit fact

The following instructions are written at a general level to help you test your understanding.

1. In My Demo Project, on the Project Tables View tab, in any project table that contains the **TOT_DOLLARS_SALES** column (such as **city_subcateg_sls** or **customer_sls**), right-click the table header and select **Create Fact**.

*The **TOT_DOLLARS_SALES** column is already used in the Revenue fact. You have to reuse this column to create the Profit fact.*

2. In the Architect window, in the box, type **Profit** as the fact name.
3. Click **OK**.
4. In the **Create New Fact Expression** window, create the following expression: **TOT_DOLLAR_SALES - TOT_COST**.
5. Under **Mapping method**, keep **Automatic** selected.
6. Click **OK**.

Create the second expression for the Profit fact

7. Edit the Profit fact.
8. In the Fact Editor, create a second expression: **QTY SOLD * (UNIT_PRICE - DISCOUNT - UNIT_COST)**.
This column can be found in the order_detail table.
9. Under **Mapping method**, keep **Automatic** selected.
10. Click **OK**.

Create the third expression for the Profit fact

11. In the Fact Editor, create a third expression: **ORDER_AMT - ORDER_COST**.
These columns can be found in the order_fact table.
12. Under **Mapping method**, keep **Automatic** selected.
13. Click **OK**.
14. In the Fact Editor, click **OK**.

Save the changes

15. On the Home tab, in the **Save** area, click **Save and Close**.
16. In the Update Schema window, ensure the following check boxes are selected:
 - Update schema logical information
 - Recalculate table keys and fact entry levels
 - Recalculate table logical sizes
17. Click **Update**.

Exercise 6.3: Modify attribute forms

As analyst use the project to create reports over time, their requirements will change. For example, you may receive requests to pull in data from a new table in the data warehouse, or to modify the report and browsing behavior for an attribute.

In this exercise, you will modify the Day and Employee attributes in My Demo Project. Create or modify the following forms and expressions for these attributes:

Attribute	Form	Expression	Source Table	Browse Form?	Report Form?
Day	ID	ORDER_DATE	order_detail	Yes	Yes
Employee	DESC	Concat ([EMP_LAST_NAME], ", ", [EMP_FIRST_NAME])	lu_employee	Yes	Yes
	First Name	EMP_FIRST_NAME		No	No
	Last Name	EMP_LAST_NAME		No	No
	SSN	EMP_SSN		Yes	No

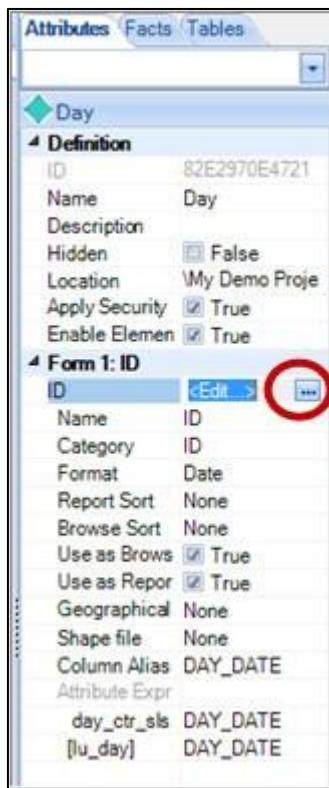
Specify the following settings for these attributes:

- Use automatic mapping for all attribute form expressions.
- Define the **Use as Browse Form** and **Use as Report Form** properties as indicated for each attribute.
- After both attributes have been modified, save and update the project schema.

Modify the Day attribute

1. In My Demo Project, in Architect, click the **Project Tables View** tab.
2. In the Layer area of the toolbar, select the **Time** layer from the drop-down list.
3. On the Project Tables View tab, in the lu_day table, click the **Day** attribute.

4. To modify the ID form for the Day attribute, in the Properties pane, under Form 1:ID, click **ID** to see the Browse button.



5. Click **Browse**.
6. In the Modify Attribute Form window, on the Definition tab, click **New**.
7. In the Create New Form Expression window, in the **Source table** drop-down list, select **order_detail**.
8. In the **Form expression** box, create the following expression: **ORDER_DATE**.
9. Under **Mapping method**, ensure **Automatic** is selected.
10. Click **OK**.
11. In the Modify Attribute Form window, click **OK**.
12. Click **Yes** on the message regarding inconsistent data type.

You do not need to configure the Use as Browse Form and Use as Report Form properties for the Day attribute because they are automatically set to True.

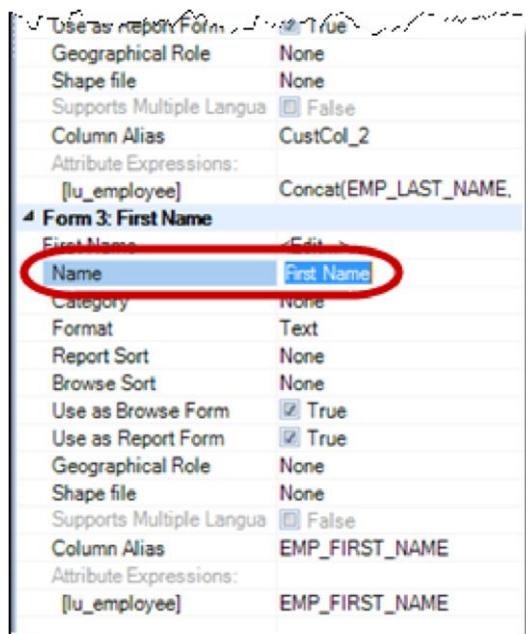
Modify the Employee attribute

Modify the Employee description (DESC) form

1. In the Layer area of the toolbar, select the **Geography** layer from the drop-down list.
2. On the Project Tables View tab, in the lu_employee table, right-click the **Employee** attribute and select **New Attribute Form**.
3. In the Create New Form Expression window, type the expression as follows:
`Concat([EMP_LAST_NAME], ", ", [EMP_FIRST_NAME])`.
4. Click **Validate** to validate the expression.
5. Under **Mapping method**, keep **Automatic** selected. **6 Click OK.**

Create the First Name form for the Employee attribute

6. Create a new attribute form for the Employee attribute with the following expression:
`EMP_FIRST_NAME`.
7. Use **Automatic mapping**.
8. In the Properties pane, modify the **Name** property of the newly created form (Form 3: None) to **First Name**.



Create the Last Name form for the Employee attribute

9. Create a new attribute form for the Employee attribute with the following expression:
EMP_LAST_NAME.
10. Use **Automatic mapping**.
11. In the Properties pane, modify the **Name** property of the newly created form (Form 4: None) to **Last Name**.

Create the SSN form for the Employee attribute

12. Create a new attribute form for the Employee attribute with the following expression: **EMP_SSN**.
13. Use **Automatic mapping**.
14. In the Properties pane, modify the **Name** property of the newly created form (Form 5: None) to **SSN**.

Define the report display and browse forms for the Employee attribute

15. In the lu_employee table, select the **Employee** attribute, if it is not already selected.
16. In the Properties pane, under Form 2: DESC, ensure that the **Use as Browse Form** and **Use as Report Form** properties are set to **True**.
17. Under Form 3: First Name, modify the **Use as Browse Form** and **Use as Report Form** properties to **False**.

To set the property to False, clear the box.

18. Under Form 4: Last Name, set the **Use as Browse Form** and **Use as Report Form** properties to **False**.
19. Under Form 5: SSN, modify the **Use as Report Form** property to **False**.
20. Keep the **Use as Browse Form** property set to **True**.
21. **Save the changes**
22. Save and close Architect and update the project schema.

Exercise 6.4: Create metrics using facts

Creating a simple metric

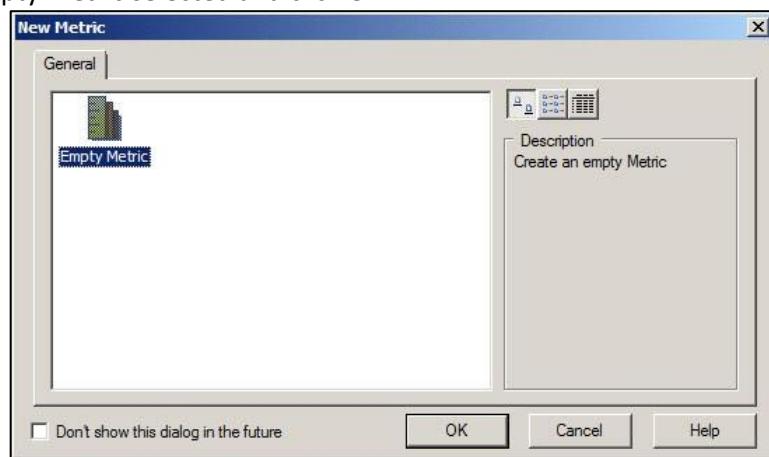
A metric is a calculation performed on stored data to help you answer a business question. For example, a metric in your project may calculate the average dollar amount of profits. A simple metric applies a function (such as sum or average) to a fact, attribute, or another metric.

You create metrics in your project to display calculated fact data on reports. The simplest way to do this is to use the Metric Creation Settings option in Architect Settings. When this option is set for a fact created in Architect, a metric with the same name as the fact is created in the Public Objects\Metrics folder.

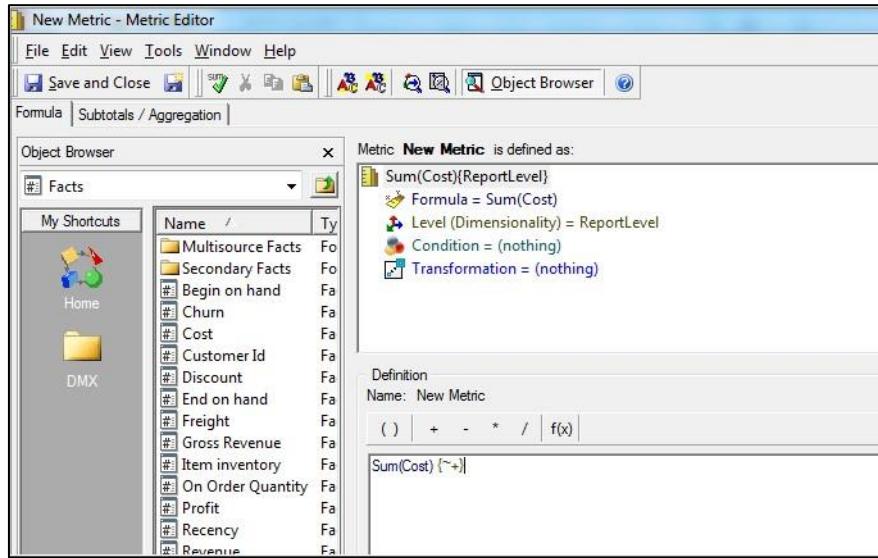
You can create a metric for each fact manually using the Metric Editor. In the following exercise, you will use this method to create a simple metric.

Create the Average Cost metric

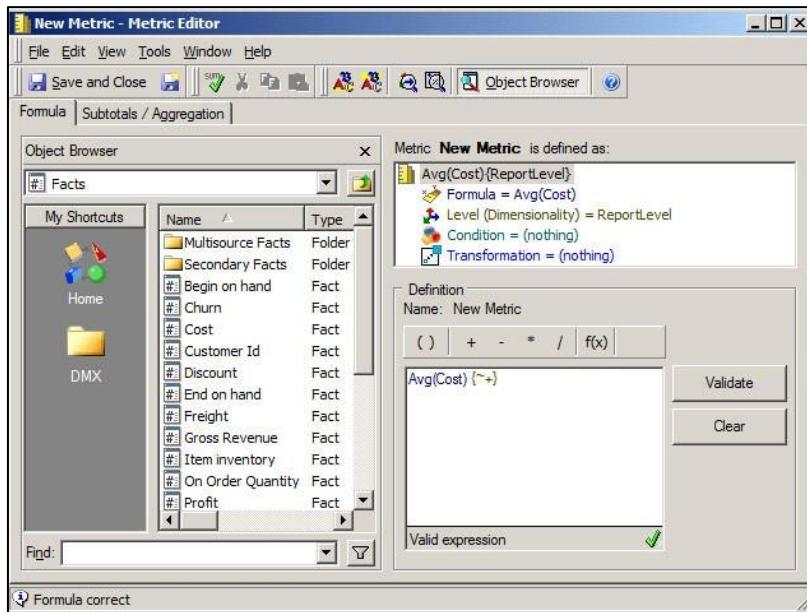
1. In MicroStrategy Developer, log in to the **MicroStrategy Analytics Modules** three-tier project source using **administrator** as your **Login ID** and leave the password textbox blank.
2. Click **OK**.
3. Expand the MicroStrategy Tutorial project.
4. Browse to the My Personal Objects\My Reports folder.
5. Right-click in the Object Viewer, point to New, and select Folder.
6. Name the new folder Project Architecting. Double-click the new folder to open it.
7. From the File menu, point to New, and select Metric.
8. Keep Empty Metric selected and click OK.



9. In the Metric Editor, under the Object Browser panel on the left, from the **Facts** folder, drag the **Cost** fact to the **Definition** box on the right. By default, the Sum function is applied to Cost.



10. To create an average cost metric, in the Definition pane, delete **Sum** and type **Avg**.



11. Click **Validate** to check your formula.

12. Click **Save and Close**. Type **Avg Cost** as the new metric's name and click **Save**.

CHAPTER 7: TRANSFORMATIONS

Transformations compare values for two different time periods. For example, this year versus last year, or today versus month to date. Transformations help analysts discover time-based trends in data. To make your MicroStrategy schema efficient, you build transformations as standalone objects and then apply them to metrics to create transformation metrics.

For example, if you apply a last year transformation to a revenue metric, the metric will display revenue values for the previous year on a report. In the following report, the Revenue metric displays revenue values for each item in the year 2012, while the Last Year's Revenue metric displays revenue for each item in the year 2011.

Current Versus Last Year's Revenue				
		Report details	x	
		Report Description:		
Report Filter: (Year = 2012) And (Subcategory = Kids / Family)				
Year	Item	Metrics	Last Year's Revenue	Revenue
2012	The Rugrats Movie	\$20,694	\$18,450	
	A Bug's Life	\$23,730	\$16,875	
	Mulan	\$21,497	\$16,711	
	Charlotte's Web	\$20,110	\$16,095	
	The Little Mermaid	\$26,097	\$20,618	
	The Lion King	\$15,579	\$12,781	
	Lady and the Tramp	\$18,660	\$13,982	
	The Jungle Book	\$20,435	\$15,501	
	Alice in Wonderland	\$16,153	\$12,286	
	Barney	\$19,470	\$15,025	
	Blue's Clues	\$13,455	\$10,265	
	The Parent Trap	\$15,216	\$12,171	
	The Muppet Movie	\$20,285	\$15,150	
	The Year Without a Santa Claus	\$17,191	\$13,326	
	Sesame Street	\$14,292	\$10,935	

Any transformation can be included as part of the metric definition, and multiple transformations can be applied to the same metric.

Types of transformations

Transformations are categorized by the methods you use to create them:

- **Table-based transformations** are created using a transformation table in the warehouse.
- **Expression-based transformations** are created using a mathematical formula.

Table-based transformations

Table-based transformations are created by defining time period relationships in tables in the data warehouse, often during the ETL process. Transformation data is incorporated into the lookup tables for attributes through transformation columns. For example, the MicroStrategy Tutorial project includes the following LU_DAY table in the data warehouse:

LU_DAY Lookup Table

DAY_DATE	PREV_DAY_DATE	LM_DAY_DATE	LQ_DAY_DATE	LY_DAY_DATE
12/30/2012	12/29/2012	11/30/2012	9/30/2012	12/30/2011
12/31/2012	12/30/2012	11/30/2012	10/1/2012	12/31/2011
1/1/2013	12/31/2012	12/1/2012	10/2/2012	1/1/2011
1/2/2013	1/1/2013	12/2/2012	10/3/2012	1/2/2011
1/3/2013	1/2/2013	12/3/2012	10/4/2012	1/3/2012
1/4/2013	1/3/2013	12/4/2012	10/5/2012	1/4/2011
1/5/2013	1/4/2013	12/5/2012	10/6/2012	1/5/2011
1/6/2013	1/5/2013	12/6/2012	10/7/2012	1/6/2011
1/7/2013	1/6/2013	12/7/2012	10/8/2012	1/7/2012
1/8/2013	1/7/2013	12/8/2012	10/9/2012	1/8/2012
1/9/2013	1/8/2013	12/9/2012	10/10/2012	1/9/2012
1/10/2013	1/9/2013	12/10/2012	10/11/2012	1/10/2012

This table also has columns for the parent IDs at all levels. These columns are not displayed in the image above.

Each date in the LU_DAY table has a transformed value for the previous day, last month's date, last quarter's date, and last year's date. For example, for January 5, 2013, the previous day was January 4, 2013, while the last quarter's date was October 5, 2012.

Other types of transformations may require separate transformation tables. For example, the following image shows the table that stores the values for the month to date transformation, the MTD_DAY table:

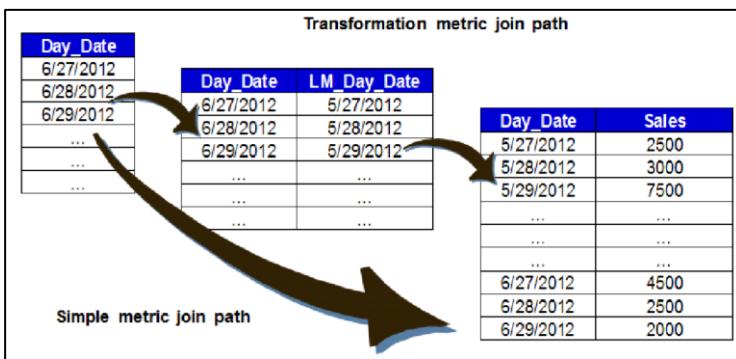
Month to Date Transformation Table	
DAY_DATE	MTD_DAY_DATE
12/31/2012	12/31/2012
1/1/2013	1/1/2013
1/2/2013	1/1/2013
1/2/2013	1/2/2013
1/3/2013	1/1/2013
1/3/2013	1/2/2013
1/3/2013	1/3/2013
1/4/2013	1/1/2013
1/4/2013	1/2/2013
1/4/2013	1/3/2013
1/4/2013	1/4/2013

In the MTD_DAY transformation table, each date can have multiple records. For example, there is only one record for the January 1, 2013 date, but there are three records for the January 3, 2013 date.

This type of transformation data cannot be stored in the lookup table for the Day attribute because lookup tables store each unique date only once.

When you create a transformation metric, the Engine uses the transformation to generate SQL for that metric. The following diagram shows how transformation tables act as intermediaries in the metric join path when you use transformation metrics on a report:

Transformation Tables in Metric Join Path



Depending on the database being used for the data warehouse, a table-based transformation may be required when performing a many-to-many transformation such as a year-to-date calculation. Table-based transformations are also required when the transformation cannot be created using an expression.

Expression-based transformations

An expression-based transformation is defined using a mathematical expression. A transformation expression typically includes an attribute ID column, a mathematical operator, and a constant.

For example, a Last Quarter or Last Month transformation can be created using QUARTER_ID-1 or MONTH_ID-1, respectively. You can also create expression-based transformations using pass-through functions such as ApplySimple. These types of expressions leverage database-specific functions to calculate transformations.

Expression-based transformations require the data to be in a format that is conducive to calculation. For example, if month IDs are stored using a YYYYMM format, the MONTH_ID-1 expression may not produce the desired results. If you apply the expression to a month ID of 201101 (January 2011), you would expect the result to be a month ID of 201012 (December 2010). However, the operation would produce an invalid month ID of 201100.

Transformation components

The following components are used to define a transformation.

Member attributes

Member attributes define the possible levels of aggregation for a transformation in a report. In other words, they define the lowest-level attribute on the report to which the transformation can be applied. For example, if the report is analyzed at the quarter level, a Quarter member attribute must be added to the transformation. If the transformation is month to date, the member attribute must be Day to achieve the desired level aggregation on a report.

Member expressions

Each member attribute has a corresponding expression that is used to generate report SQL queries. For expression-based transformations, the member expression is a mathematical expression. For table-based transformations, it is a column in a transformation table.

A single transformation can use a combination of table-based and expression-based definitions. For example, you may create a Last Year transformation based on the Year, Month, and Day attributes. Year may be defined with an expression such as YEAR_ID-1, while Month and Day must be mapped to columns in a transformation table because their values are not conducive to expressions.

Member tables

The member tables store the data for the member attributes. For expression-based transformations, the member tables are generally lookup tables that correspond to the attribute being transformed, such as LU_DAY, LU_QUARTER, and so on. For table-based transformations, it is the transformation table that stores the relationship.

Mapping type

The mapping type identifies the transformation cardinality. The mapping type can be one of the following:

- **One-to-one:** A typical one-to-one relationship is last year versus this year. A year maps to exactly one other year.
- **Many-to-many:** A typical many-to-many relationship is year to date. A date maps to itself in addition to every date that came before it in the given year.

Exercises

Exercise 7.1: Create a last year's transformation

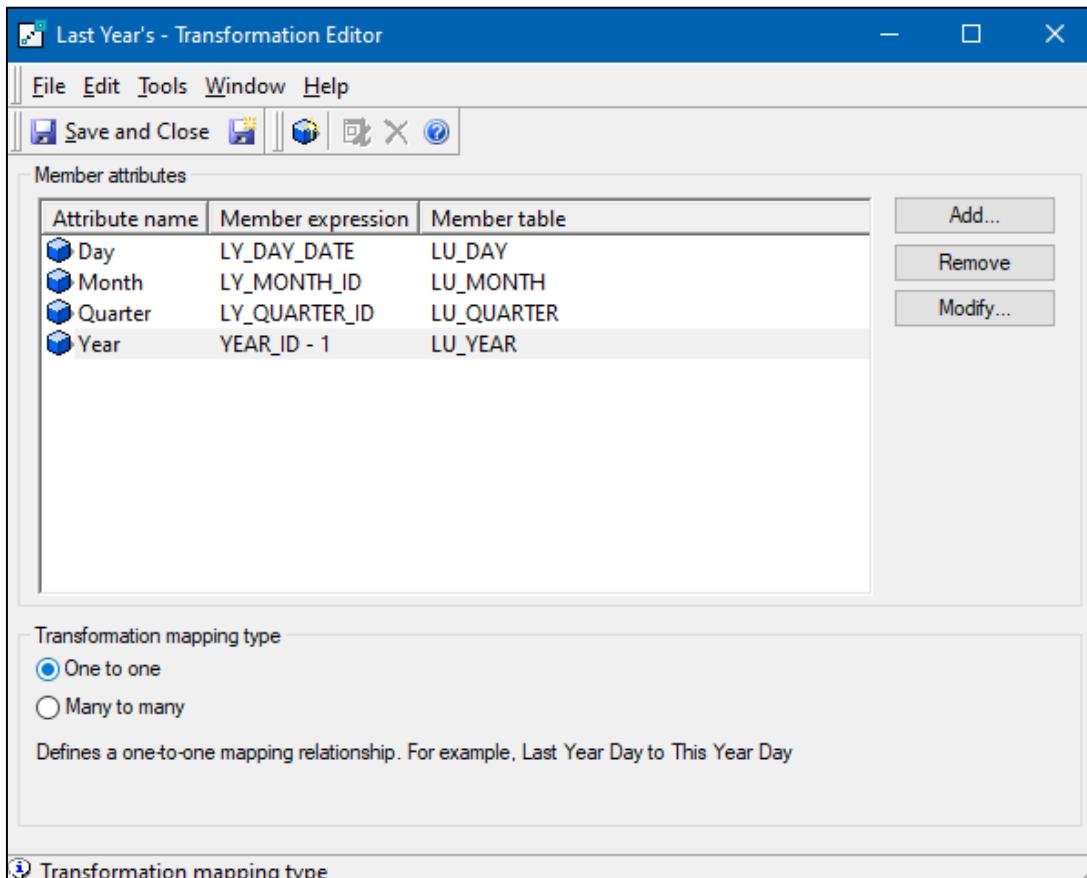
In this exercise, you will create a Last Year's transformation in the **MicroStrategy Tutorial Project**. The transformation definition includes a Year member attribute with a [YEAR_ID] - 1 expression defined on the LU_YEAR table.

To apply the transformation, you will create the Last Year's Revenue transformation metric.

Next, you will create a report that compares the Revenue and Last Year's Revenue metrics:

Report View: 'Local Template'			
Year	Metrics	Revenue	Last Year's Revenue

When you add the Quarter attribute to this report, you will notice that your transformation metric does not aggregate values to the desired level. To correct this issue, you will add additional member attributes to the Last Year's transformation:



To test the modifications to your transformation, you will run the Transformation Example report and drill down to a finer level of detail.

Create the transformation

1. In Developer, open the **MicroStrategy Tutorial Project**.

*If you were logged out, you can log in to **MicroStrategy Analytics Modules Project Source** using the following credentials:*

- login id: administrator
- password: (none)

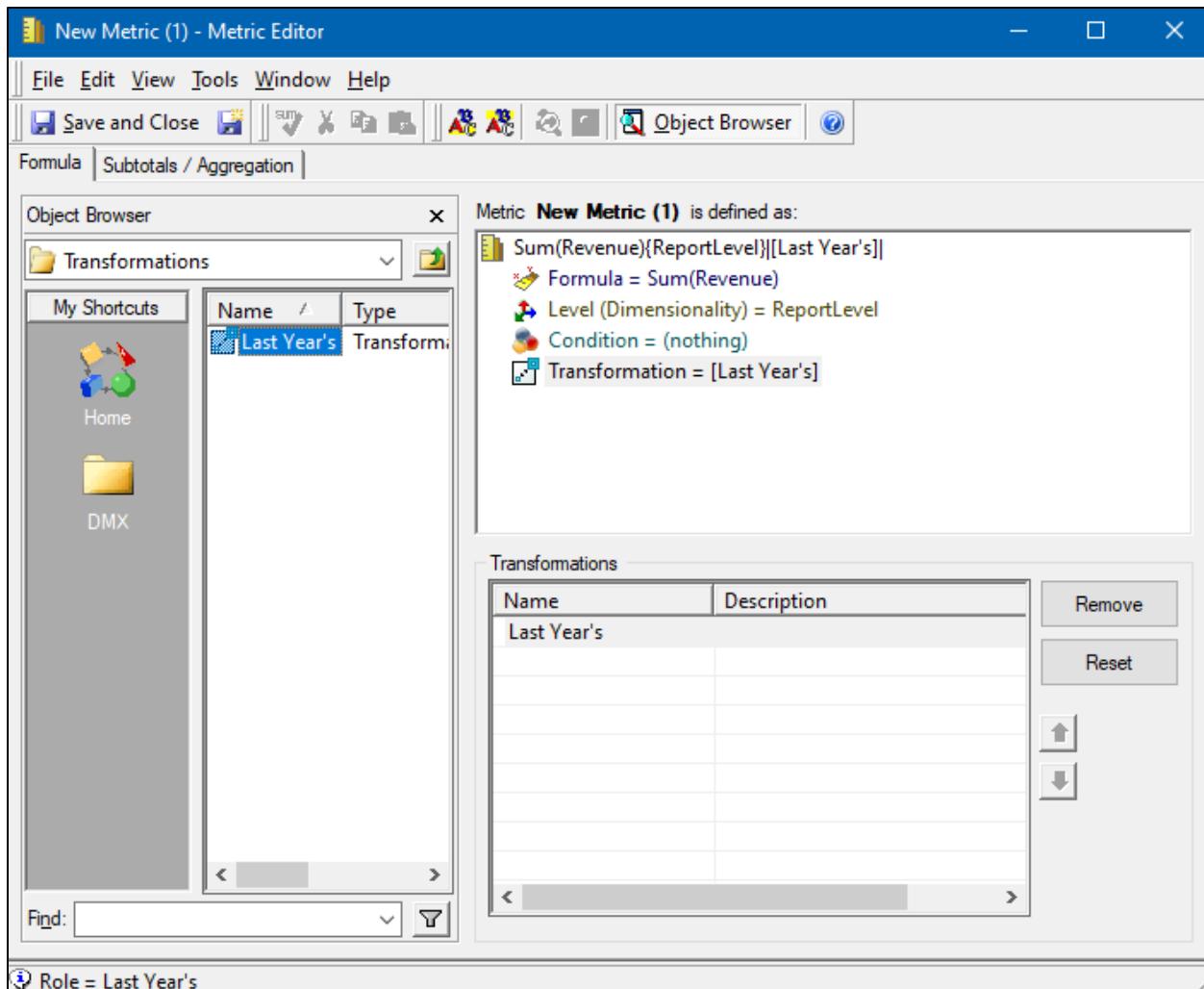
2. In the project, expand the **My Personal Objects** folder.

3. Click the **My Reports** folder.
 4. Create a new folder name **Transformations**.
 5. From the **File** menu, point to **New** and select **Transformation**.
 6. In the Select a Member Attribute window, double-click the **Time** folder and select **Year**.
 7. Click **Open**.
 8. In the Define a new member attribute expression window, in the **Table** drop-down list, select the **LU_YEAR** table.
 9. In the Member attribute expression pane, type **[YEAR_ID] - 1**.
 10. Click **Validate**, and then click **OK**.
 11. Click **Save and Close**.
 12. Save the transformation in the **My Personal Objects\My Reports\Transformations** folder as **Last Year's**.
 13. Update the project schema. To do this, click the **Update Schema** button on the toolbar.
-

Create a transformation metric that uses the new transformation

14. In the My Personal Objects folder, in the My Reports folder.
15. Right-click in the Object Viewer, point to New, and select Folder.
16. Name the new folder Transformation Metric. Double-click the new folder to open it
17. From the File menu, point to New, and select Metric.
18. Keep Empty Metric selected and click OK.
19. Define the metric as Sum(Revenue).
20. From the Tools menu of the Metric Editor, point to Formatting and select Values. Click Currency and specify 0 decimal places.
21. In the Metric: New Metric is Defined As pane, click Transformation = (nothing).
22. In the Object Browser, browse to the Last Year's transformation you created above.

23. Your metric should resemble the following:



24. Save the metric in the **My Personal Objects\My Reports\Transformation Metrics** folder as **Last Year's Revenue**. Close the metric.

Create a report with the new transformation metric

1. In the **My Personal Objects\My Reports** folder, create the following report:

Report View: 'Local Template'			
Year	Metrics	Revenue	Last Year's Revenue

- a. You can find the Year attribute in the Schema Objects\Attributes\Time folder.
 - b. You can find the Revenue metric in the Public Objects\Metrics folder.
 - c. You can find the Last Year's Revenue metric in the My Personal Objects\My Reports\Transformation Metrics folder.
2. Run the report. The report result set should resemble the following:
- | Year | Metrics | Revenue | Last Year's Revenue |
|------|--------------|------------|---------------------|
| | | | |
| 2015 | \$11,517,606 | 8,647,238 | |
| 2016 | \$14,858,864 | 11,517,606 | |
- 3. Only data for the 2015 and 2016 years are displayed, even though there is Revenue data for 2014 in the data warehouse. Notice that the metrics return different values for each row, but the Revenue data for 2015 is identical to the Last Year's Revenue for 2016.
- Modify the report**
- 4. Switch to Design View.
 - 5. From the Schema Objects\Attributes\Time folder, add the **Quarter** attribute to the report template to the right of Year.
 - 6. From the **Data** menu, select **Subtotals**.
 - 7. In the Subtotals window, on the Definition tab, select **Total** and then click **OK** to place the Total subtotal on the report.
 - 8. Run the report. The report result set should resemble the following:

Year	Quarter	Metrics		Last Year's Revenue
			Revenue	
2014	2014 Q1		\$1,682,656	1,682,656
	2014 Q2		\$1,985,788	1,985,788
	2014 Q3		\$2,314,295	2,314,295
	2014 Q4		\$2,664,500	2,664,500
	Total		\$8,647,238	8,647,238
2015	2015 Q1		\$2,498,756	2,498,756
	2015 Q2		\$2,684,764	2,684,764
	2015 Q3		\$3,067,019	3,067,019
	2015 Q4		\$3,267,067	3,267,067
	Total		\$11,517,606	11,517,606
2016	2016 Q1		\$3,111,989	3,111,989
	2016 Q2		\$3,504,479	3,504,479
	2016 Q3		\$3,729,456	3,729,456
	2016 Q4		\$4,512,940	4,512,940
	Total		\$14,858,864	14,858,864
Total			\$35,023,708	35,023,708

9. Because the Last Year's transformation is not defined for the Quarter attribute, the transformation metric is not evaluated correctly. The metric values are identical for both metrics. You will fix this problem in the next set of steps by adding a member attribute to the transformation.
10. Save the report in the **My Personal Objects\My Reports** folder as **Transformation Example**. Close the report.

Add the Quarter member attribute to the Last Year's transformation

1. In the **My Personal Objects\My Reports** folder, open the Transformations folder and double-click the **Last Year's** transformation.
2. In the Read Only window, select **Edit** and click **OK**.
3. In the Transformation Editor, click **Add**.
4. In the Select a Member Attribute window, navigate to the **Schema Objects\ Attributes\Time** folder and select **Quarter**.
5. Click **Open**.
6. In the Define a New Member Attribute Expression window, from the **Table** drop-down list, select the **LU_QUARTER** table.

7. Double-click the **LY_QUARTER_ID** column to move it to the Member attribute expression pane.
8. Click **OK**.

Add the Month member attribute to the Last Year's transformation

9. In the Transformation Editor, click **Add**.
10. In the Select a Member Attribute window, navigate to the **Schema Objects\ Attributes\Time** folder and select **Month**.
11. Click **Open**.
12. In the Define a New Member Attribute Expression window, from the **Table** drop-down list, select the **LU_MONTH** table.
13. Double-click the **LY_MONTH_ID** column to move it to the Member Attribute Expression pane.

14. Click **OK**.

Add the Day member attribute to the Last Year's transformation

15. In the Transformation Editor, click **Add**.
16. In the Select a Member Attribute window, navigate to the **Schema Objects\ Attributes\Time** folder and select **Day**.
17. Click **Open**.
18. In the Define a New Member Attribute Expression window, from the **Table** drop-down list, select the **LU_DAY_DATE** table.
19. Double-click the **LY_DAY_DATE** column to move it to the Member Attribute Expression pane.
20. Click **OK**.
21. Save and close the transformation.
22. Update the project schema.
23. **Run the transformation report**
24. Run the Transformation Example report you saved in the **My Personal Objects\My Reports** folder. The report result set resemble the following:

Year	Quarter	Metrics	Last Year's Revenue	
			Revenue	
2015	2015 Q1		\$2,498,756	1,682,656
	2015 Q2		\$2,684,764	1,985,788
	2015 Q3		\$3,067,019	2,314,295
	2015 Q4		\$3,267,067	2,664,500
	Total		\$11,517,606	8,647,238
2016	2016 Q1		\$3,111,989	2,498,756
	2016 Q2		\$3,504,479	2,684,764
	2016 Q3		\$3,729,456	3,067,019
	2016 Q4		\$4,512,940	3,267,067
	Total		\$14,858,864	11,517,606
Total			\$26,376,470	20,164,844

This result set is correct with the quarter-level transformation. The Revenue total for 2015 is identical to the Last Year's Revenue total for 2016, as expected.

25. Right-click the **2015 Q1** quarter element, point to **Drill**, point to **Down**, and select **Day**. The drill-down report result set should resemble the following:

Year	Quarter	Day	Metrics	Last Year's Revenue	
				Revenue	
		1/1/2015		\$24,117	13,475
		1/2/2015		\$23,530	11,689
		1/3/2015		\$16,714	17,406
		1/4/2015		\$17,282	10,287
		1/5/2015		\$18,939	14,595
		1/6/2015		\$33,039	22,501
		1/7/2015		\$34,815	25,289
		1/8/2015		\$18,678	23,993
		1/9/2015		\$21,399	15,613
		1/10/2015		\$20,122	15,845
		1/11/2015		\$35,040	12,955
		1/12/2015		\$29,461	14,755
		1/13/2015		\$27,487	10,173
		1/14/2015		\$23,464	15,322
		1/15/2015		\$13,051	15,291
		1/16/2015		\$19,173	12,378
		1/17/2015		\$20,432	18,320
		1/18/2015		\$22,945	18,585
		1/19/2015		\$32,276	18,246

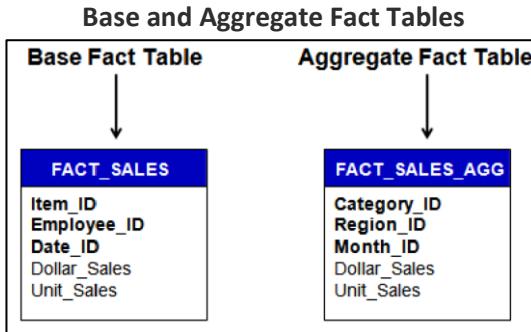
The image above only displays the first few rows of the result set for the report.

26. Close both reports without saving.

CHAPTER 8: AGGREGATE TABLES

Base fact tables store fact data at the lowest possible level of detail, while aggregate fact tables store fact data at a higher, or summarized, level of detail.

For example, consider the following two fact tables:



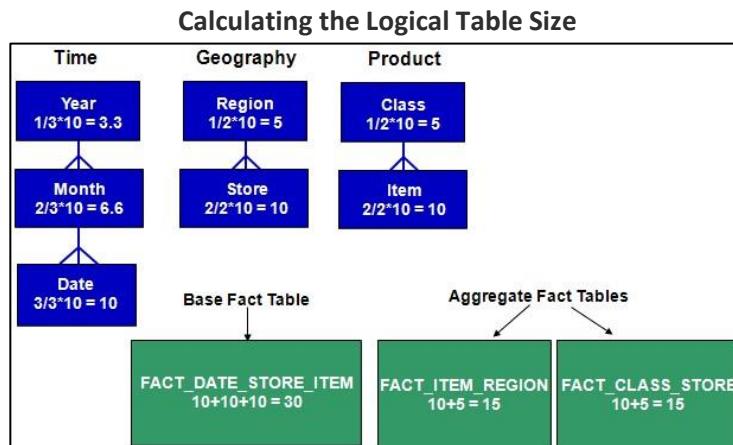
Because they store data at a higher level than base tables, aggregate fact tables reduce query time. For example, when users run a report that shows unit sales by region, they will obtain a result set more quickly if the report uses the FACT_SALES_ AGG table rather than the FACT_SALES table.

Logical table size

Architect assigns a logical size to every table that is added to a project, based on the table's columns and mapped attributes. When the SQL Engine can retrieve data from two or more tables in the warehouse, it inspects the logical table sizes and generates SQL against the table with the smallest logical table size. This process helps the SQL Engine select the optimal table for a query.

You can view the logical table size for each table in the Logical Table Editor.

The following diagram is a visual representation of the algorithm used by Architect to assign logical table sizes:



Logical table size is the sum of the weight for each attribute contained in the table. Attribute weight is defined as the position of an attribute in its hierarchy divided by the number of attributes in the hierarchy, multiplied

by a factor of 10. Using this formula, Architect calculates the respective weight of each attribute as shown in the diagram above. The logical table size of each fact table is simply the sum of its respective attribute weights.

Changing the logical table size

If you want to control the MicroStrategy Engine's preference for a given table, you can modify that table's logical table size. For example, in the diagram above, there are two aggregate fact tables that both have the same logical table size of 15. One of these tables contains item and region information, while the other has class and store information. Because there are many more items than item classes, the table with item and region information is much larger. To force the Engine to account for this size disparity, you can modify the logical table sizes.

In general, small physical tables are assigned a small logical table size. Tables with higher-level attributes usually have a smaller logical table size than tables with lower-level attributes. However, there may be cases where you want to customize the Engine's query behavior. To do this, you can modify the logical table size to force the SQL Engine to use the table that you know has a smaller physical size.

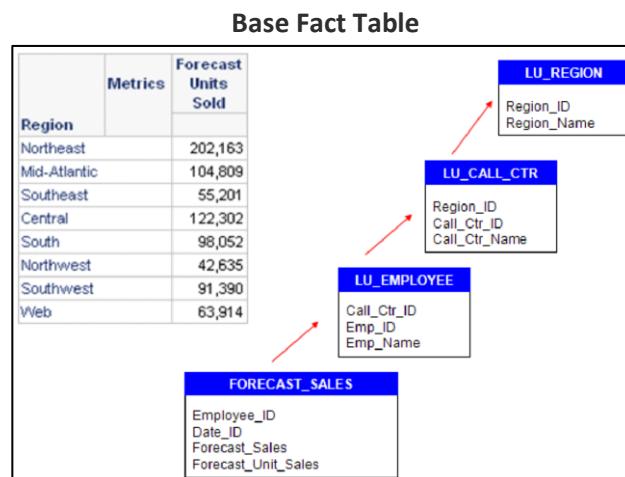
Data mart

A data mart is a relational table that contains a report's result set. After you create a data mart, you can use it as a source table in your project and execute reports against it. Common use cases for data marts include:

- Creating aggregate fact tables
- Creating tables for very large result sets and then using other applications such as Microsoft Excel or Microsoft Access to access this data
- Creating tables for off-line analysis

In this lesson, you will use data marts to create aggregate fact tables.

For example, consider the following scenario:



In this example, forecasting data is stored at the employee and date level in the FORECAST_SALES base fact table. However, you want to report on the Forecast Units Sold at the Region level. This requires three joins from the fact table to the LU_REGION lookup table.

This query may be very costly, especially if users run it often.

What if you could create an aggregate table that limits the number of joins and the number of rows in the fact table? This can be achieved by creating a data mart table. You can bring this table into a project, map the Forecast Unit Sales metric to it, and develop region-level reports that automatically use it, as shown below:

Aggregate Fact Table Created as Data Mart



Data mart objects

A data mart consists of the following:

- **Data mart report:** A metadata object created in the Report Editor. When executed, the data mart report creates the data mart table in the warehouse of your choice. The data mart report contains attributes, metrics, and other application objects that translate into columns in the data mart table.
- **Data mart table:** The relational table created after the execution of a data mart report.

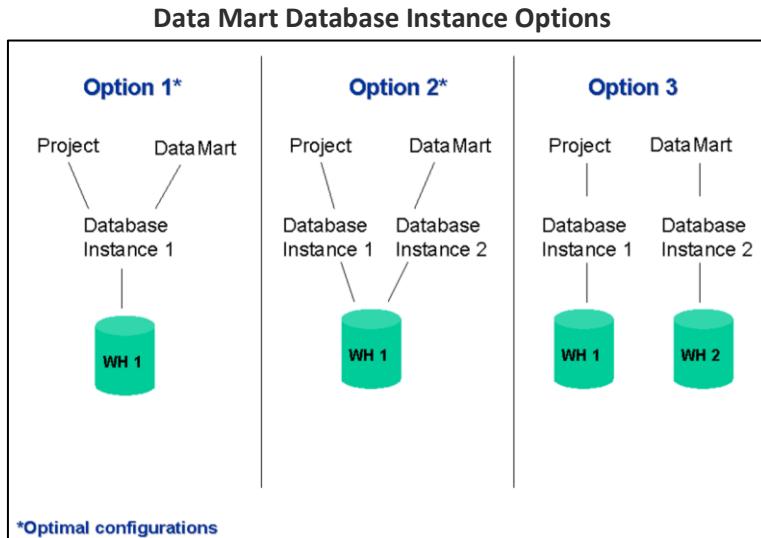
Data mart database instances

When you create a data mart report, you must specify a database instance in which to create the data mart table.

Create a data mart in a database instance in one of the following ways:

- Option 1: Use the project's primary database instance.
- Option 2: Use a secondary project database instance that exists in the same warehouse as the primary project database instance.
- Option 3: Use a different database instance in a different warehouse than the primary project database instance.

The following diagram shows each of these data mart database instance options:



If the primary project database instance is used, there is no need to take any additional steps to create a data mart. Simply select the primary data mart database instance as a target when you create the data mart report.

If you want to use a secondary project database instance, you must create that database instance before creating the data mart. This database can then associate the instance to the project in the Project Configuration Editor.

Exercises

Exercise 8.1: Create aggregate fact table

An aggregate fact table stores fact data at a summarized level of detail to decrease the number of joins required to return data. To create an aggregate table in your project, you can develop a data mart, which is a relational table that contains a report's result set.

Summary of steps to create an aggregate table using a data mart

In this exercise, you will develop a new aggregate table by creating a data mart. You will bring the new table into the project and modify its logical table size to control how the MicroStrategy Engine targets tables.

Before you create the data mart, you will specify a custom column alias for the Forecast Revenue metric. This allows you to insert familiar column names for temporary tables in the generated SQL.

In the next step, you will create a data mart report with Region ID and Quarter ID attribute forms and the Forecast Revenue and Forecast Units Sold metrics.

You will then add the new aggregate table to the project, and define new attribute expressions to enable facts to use the table.

Finally, you will change the logical table size for the new aggregate table and run the report to see how your changes impact the report SQL.

Creating an aggregate table using a data mart

Create the metrics

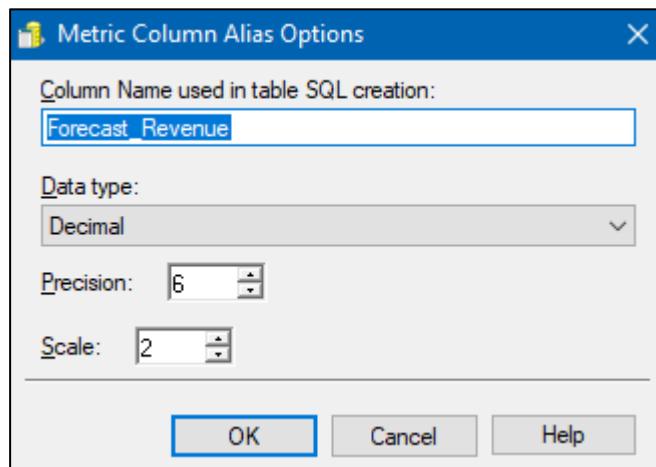
1. In Developer, open the **Forecasting Project**.

*If you were logged out, you can log in to **MicroStrategy Analytics Modules Project Source** using the following credentials:*

- login id: administrator
- password: (none)

Edit a metric to change the column name used in the SQL

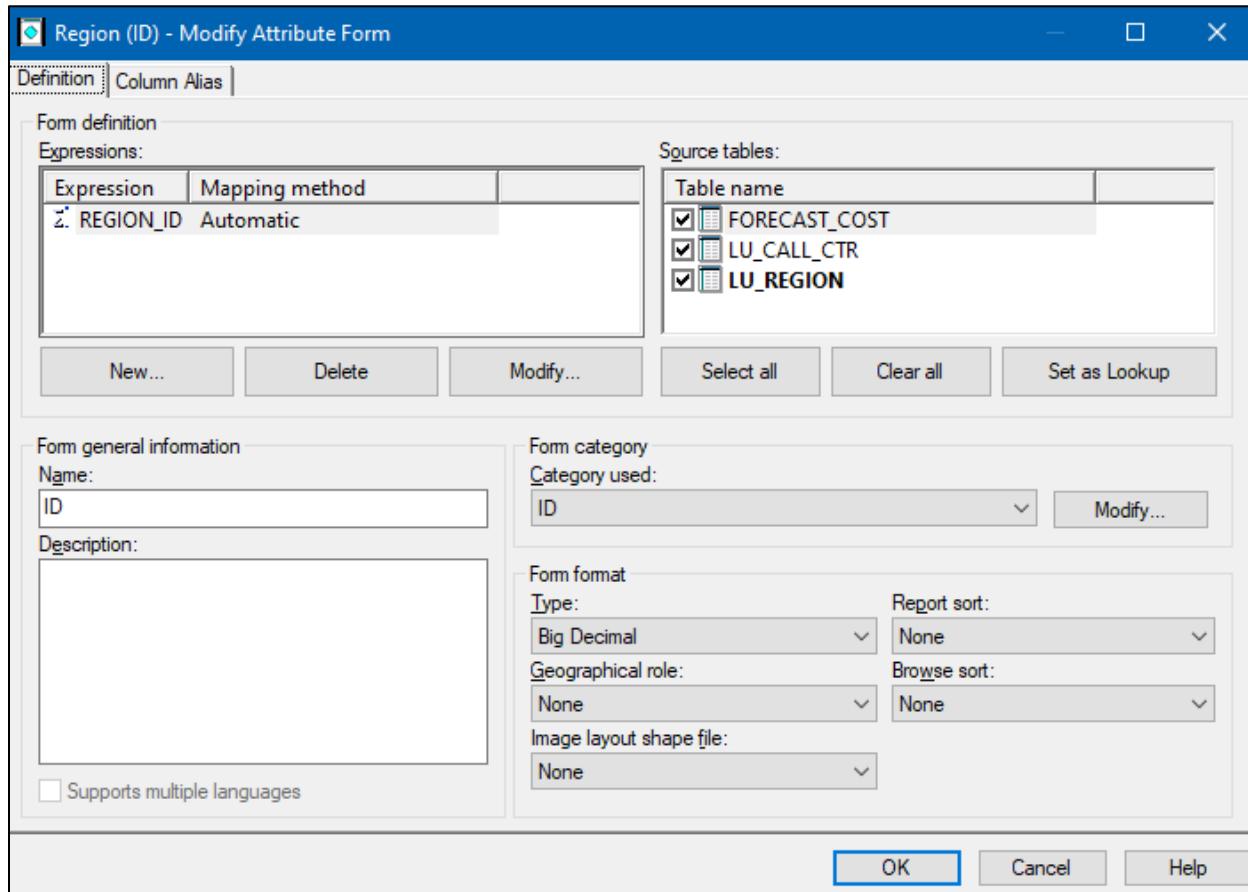
2. In the Public Objects\Metrics\Forecasting folder, double-click the **Forecast Revenue** metric to edit it.
3. In the Metric Editor, on the **Tools** menu, point to **Advanced Settings** and select **Metric Column Options**.
4. In the Metric Column Alias Options window, in the **Column Name used in table SQL creation** box, type **Forecast_Revenue**.
5. In the Data type drop down, select decimal.



6. Click **OK** to close the Metric Column Alias Options window.
7. Save and close the metric.

8. Repeat the steps for the Forecast Units Sold metric to update the column alias to **Total_Units_Sales**.

Note: You also need to modify the Region and Quarter attribute to change the data type of their ID forms to avoid any errors while configuring the data mart report and update the schema.



Create the data mart report

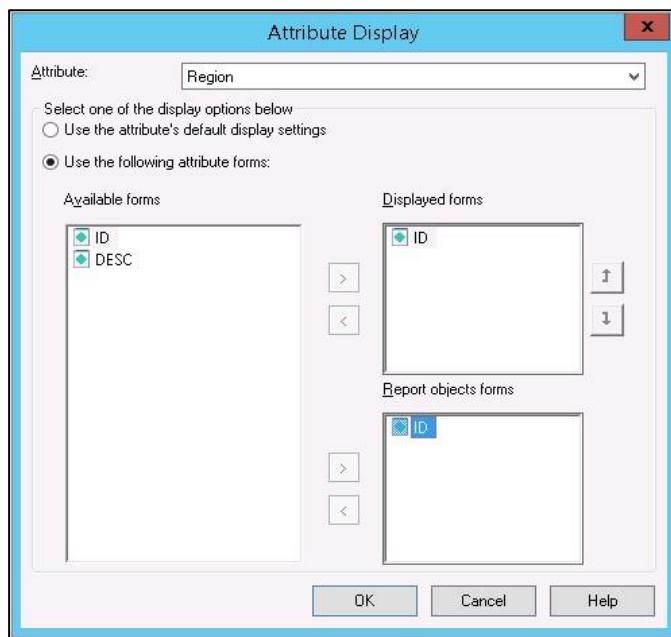
1. In the Public Objects folder, in the Reports folder, create the following report:

Report View: 'Local Template'			Switch to:
	Metrics	Forecast Revenue	Forecast Units Sold
Region	Quarter		

2. You can access the **Region** and **Quarter** attributes from the Schema Objects\Attributes\Forecasting folder.
3. The **Forecast Revenue** and **Forecast Units Sold** metrics are located in the **Public Objects\Metrics\Forecasting** folder.

Configure the attribute display options

4. In the Report Editor, from the **Data** menu, select **Attribute Display**.
5. In the Attribute Display window, from the **Attribute** drop-down list, select **Region** and then specify the following display settings:
 - a. Under **Select one of the display options below**, click **Use the following attribute forms**.
 - b. In the **Available forms** list, click the **ID** form.
 - c. Click the upper **>** button to move the ID form to the **Displayed Forms** list.
 - d. In the **Displayed Forms** list, click the **DESC** form.
 - e. Click the upper **<** button to remove the DESC form from the **Displayed Forms** list.
 - f. In the **Report Objects Forms** list, click the **DESC** form.
 - g. Click the lower **<** button to remove the DESC form from the **Report Object**

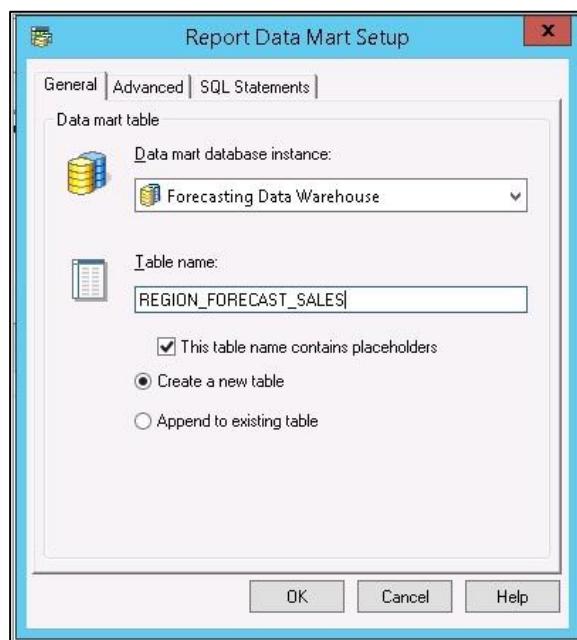


6. In the Attribute Display window, in the **Attribute** drop-down list, select **Quarter** and then specify the following display settings:

- h. Under **Select one of the display options below**, click **Use the following attribute forms**.
 - i. In the **Available forms** list, click the **ID** form.
 - j. Click the upper **>** button to move the ID form to the **Displayed Forms** list. In the **Displayed Forms** list, click the **DESC** form.
 - k. Click the upper **<** button to remove the DESC form from the **Displayed Forms** list.
 - l. In the **Report Objects Forms** list, click the **DESC** form.
 - m. Click the lower **<** button to remove the DESC form from the **Report Objects Forms** list.
7. Click **OK**.

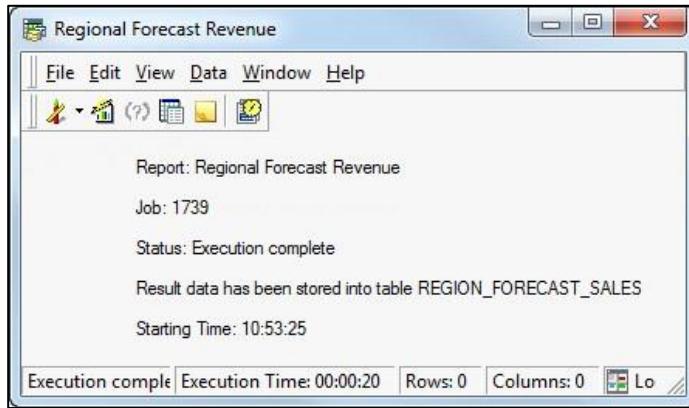
Configure the data mart

8. In the Report Editor, from the Data menu, select **Configure Data Mart**.
9. In the Report Data Mart Setup window, on the General tab, from the Data mart database instance drop-down list, select **Forecasting Data Warehouse**.
10. In the Table name box, type **REGION_FORECAST_SALES**.
11. Select **Create a new table**.



12. Click **OK**.

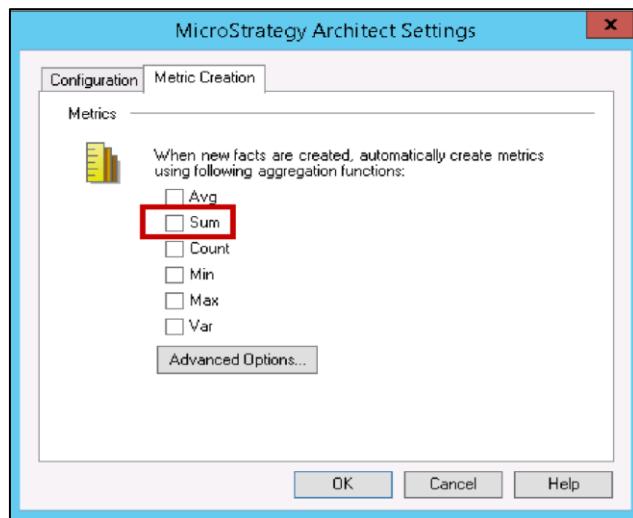
13. In the Report Data Mart Setup message window, click OK.
14. Save the report in the Public Objects\Reports folder as Regional Forecast Revenue.
15. Run the report.
16. After the report executes, the result is stored in the REGION_FORECAST_SALES table, as shown below:



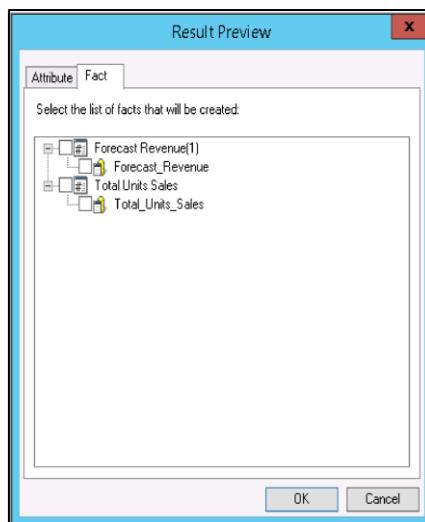
17. Close the report.

Incorporate the data mart table into the project

1. In Developer, on the **Schema** menu, select **Architect**.
2. If the Read Only message window is displayed, select **Edit: This will lock all schema objects in this project from other users.**
3. Click the **Project Tables View** tab.
4. Click the **Architect** icon and then select **Settings**.
5. In the MicroStrategy Architect Settings window, on the Metric Creation tab, clear the **Sum** check box.



6. Click **OK**. Metrics are no longer automatically created when you create new facts.
7. In the left pane, expand the **Forecasting Data Warehouse** database instance.
8. Right-click the **REGION_FORECAST_SALES** table, and select **Add Table to Project**.
9. In the Results Preview window, on the Fact tab, clear the **Forecast Revenue(1)** and **Total Unit Sales** check boxes.



10. Click **OK**.

Update the Forecast Revenue fact

11. On the Project Tables View tab, find the **FORECAST_SALES** table and select the **Forecast Revenue** fact.
12. *You may need to scroll to view the FORECAST_SALES table.*

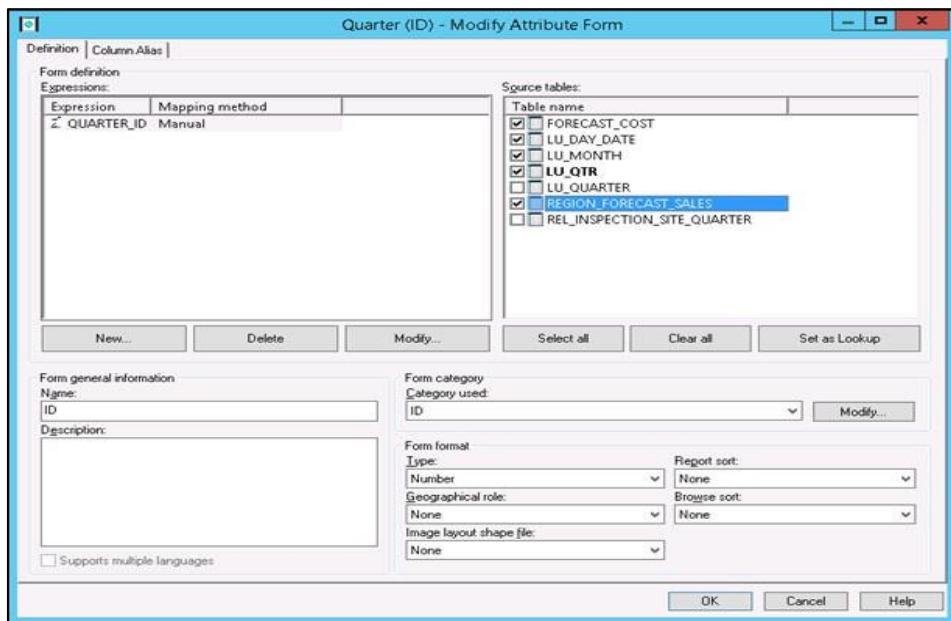
13. Right-click the **Forecast Revenue** fact and select **Edit**.
 14. In the Fact Editor, create a new fact expression that uses the Forecast_Revenue column in the REGION_FORECAST_SALES table as a source table.
 15. Under **Mapping method**, ensure **Automatic** is selected.
 16. Click **OK**, then click **OK** again to close the Fact Editor.
-

Update the Forecast Units Sold fact

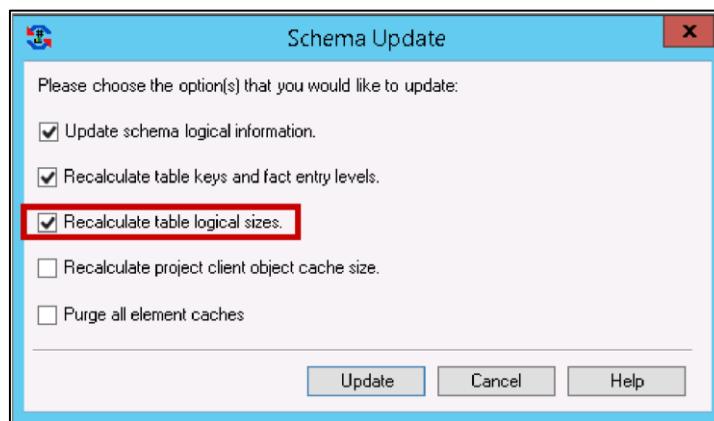
1. In the Project Tables View tab, find the **FORECAST_SALES** table and select the **Forecast Units Sold** fact.
 2. Right-click the **Forecast Units Sold** fact and select **Edit**.
 3. In the Fact Editor, create a new fact expression that uses the Total_Unit_Sales column in the REGION_FORECAST_SALES table as a source table.
 4. Under **Mapping method**, ensure **Automatic** is selected.
 5. Click **OK**, then click **OK** again to close the Fact Editor.
 6. Save and update the project schema. Then exit Architect.
-

Add REGION_FORECAST_SALES as a source table

1. In the Schema Objects\Attributes\Forecasting folder, double-click the **Quarter** attribute to edit it.
2. In the Attribute Forms window, select **ID** and click **Modify**.
3. In the Modify Attribute Form window, in the Source tables area, select the **REGION_FORECAST_SALES** table and click **OK**.



4. In the Attribute Editor, click **Save and Close**.
5. Repeat the steps for the Region attribute.
6. Save and update the project schema. Ensure that the **Recalculate table logical sizes** check box is selected in the Schema Update window.



Test the new fact mapping behavior

1. In the Public Objects\Reports folder, create the following report:

Report View: 'Local Template'				Switch to:
Region	Metrics	Forecast Revenue	Forecast Units Sold	

- a. Access the Region attribute from the Schema Objects\Attributes\ Forecasting folder.
- b. Both metrics are located in the Public Objects\Metrics\Forecasting folder.
2. Run the report. The result set should resemble the following:

Region	Metrics	Forecast Revenue	Forecast Units Sold
Northeast	13,721,807	202,163	
Mid-Atlantic	27,965,682	104,809	
Southeast	3,787,412	55,201	
Central	8,338,062	122,302	
South	6,735,528	98,052	
Northwest	11,358,447	42,635	
Southwest	6,248,587	91,390	
Web	4,321,012	63,914	

3. View the report in SQL View. The SQL should look like the following:

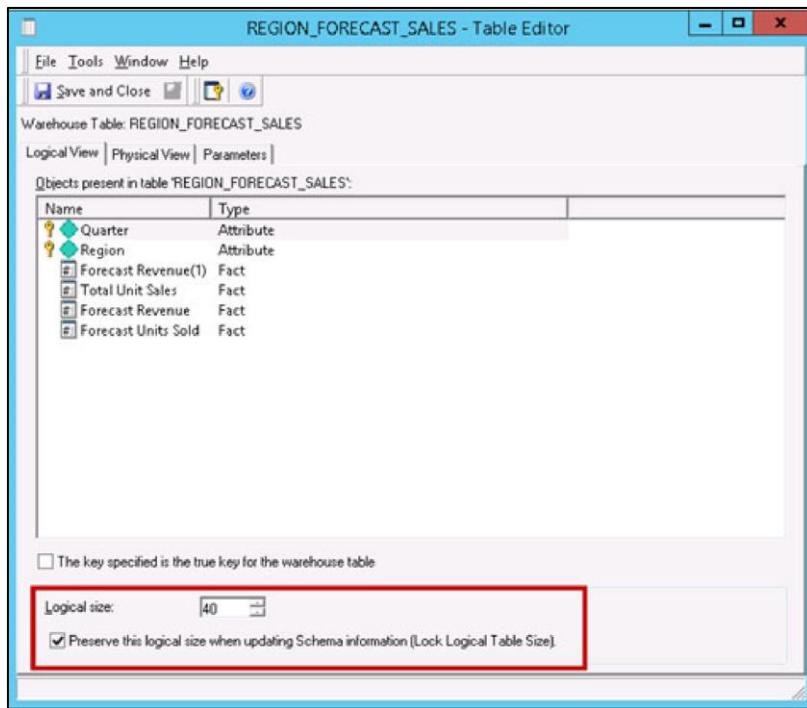
```
select a11.REGION_ID REGION_ID,
       max(a12.REGION_NAME) REGION_NAME,
       sum(a11.Forecast_Revenue) Forecast_Revenue,
       sum(a11.Total_Unit_Sales) Total_Unit_Sales
  from `REGION_FORECAST_SALES` a11
  join `LU_REGION` a12
    on (a11.REGION_ID = a12.REGION_ID)
 group by a11.REGION_ID
```

4. In the FROM clause, notice that the data is retrieved from the new aggregate table REGION_FORECAST_SALES.
5. Save the report in the **Public Objects/Reports** folder as **Data Mart Test**.
6. Close the report.

Change the logical table size for the aggregate table

1. In the Schema Objects\Tables\Forecasting folder, double-click **FORECAST_SALES** to open the Table Editor.
2. Notice that the logical size for the FORECAST_SALES table is 20.
3. Close the FORECAST_SALES Table Editor.

4. In the Schema Objects\Tables\ folder, double-click **REGION_FORECAST_SALES**.
5. Notice that the logical size for the REGION_FORECAST_SALES table is lower than that of the FORECAST_SALES table. When you execute any report that aggregates the forecast revenue data to the region or quarter level, the Engine chooses the REGION_FORECAST_SALES table because of its lower logical table size.
6. Increase the **Logical size** value to **40** and select the check box for **Preserve**
7. **This Logical Size When Updating Schema Information (Lock Logical Table Size)**. When you select this check box, the logical size for the table will not be recalculated when you update the schema.



8. Click **Save and Close**.
9. Update the project schema. Ensure that the **Recalculate table logical sizes** check box is selected in the Schema Update window.

Test the change of the logical table size on the report SQL

10. In the Public Objects\Reports folder, right-click the **Data Mart Test** report and select **View SQL**. The SQL should resemble the following:

```

select    a13.REGION_ID REGION_ID,
          max(a14.REGION_NAME) REGION_NAME,
          sum((a11.FORECAST_QTY_SOLD * (a11.FORECAST_UNIT_PRICE - a11.FORECAST_DISCOUNT))) Forecast_Revenue,
          sum(a11.FORECAST_QTY_SOLD) Total_Units_Sales
from      'FORECAST_SALES'    a11
join      'LU_EMPLOYEE'     a12
        on      (a11.EMP_ID = a12.EMP_ID)
join      'LU_CALL_CTR'     a13
        on      (a12.CALL_CTR_ID = a13.CALL_CTR_ID and
a12.COUNTRY_ID = a13.COUNTRY_ID)
join      'LU_REGION'       a14
        on      (a12.COUNTRY_ID = a14.COUNTRY_ID and
a13.REGION_ID = a14.REGION_ID)
group by  a13.REGION_ID

```

11. Notice that the Engine now retrieves data from the base FORECAST_SALES table over the aggregate REGION_FORECAST_SALES table because it has a smaller logical table size.
12. Close the report without saving it.

CHAPTER 9: MICROSTRATEGY MULTISOURCE OPTION

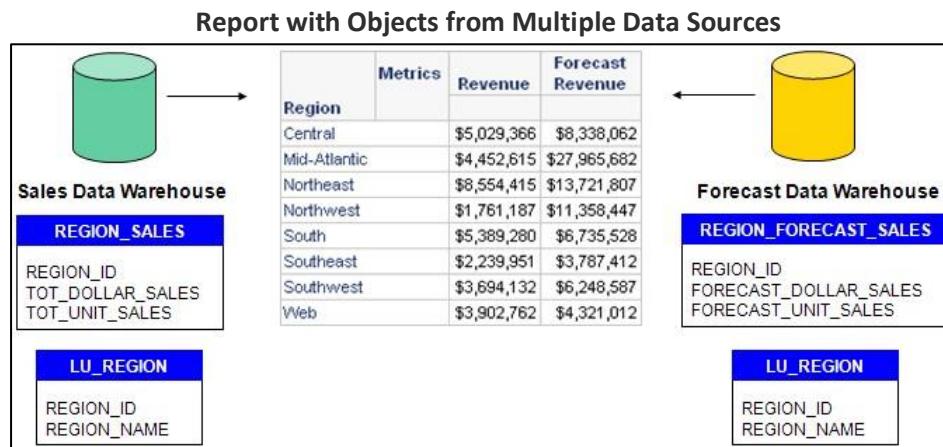
The MultiSource option enables you to connect a project to multiple data sources. This allows users to perform reporting tasks in a single project, regardless of how data is physically distributed across multiple data warehouses.

In this lesson, you will use the MultiSource Option to create reports that display data from multiple data sources.

MultiSource Option

By default, the objects in a standard report are retrieved from a single data source. If your data is distributed across multiple data warehouses, you can use the MultiSource Option—an add-on component to Intelligence Server—to define a single project schema that retrieves data from multiple data sources.

For example, consider the following scenario in which actual revenue data is stored in one data warehouse, while forecast revenue data is stored in a second data warehouse:



In this example, if you want to create a report that includes the revenue and forecast revenue data for each region, you must execute SQL against both data warehouses to retrieve the result set. The data for each metric must be retrieved from its respective data warehouse, while the region data can be obtained from either data warehouse. To retrieve data for this report from multiple sources, you can use the MultiSource Option.

Designating primary and secondary database instances

Primary and secondary database instances are designated at the project level. With the MultiSource Option, you can also define primary and secondary database instances at the table level. This capability enables you to define the project schema using multiple relational data sources.

After you select a primary database instance for the project, you can perform the following tasks to specify multiple sources in the project schema:

- Add tables to the project from various database instances, not just the primary database instance for the project.
Any SQL database instance that exists within the project source can be used as source for project select tables.
- You can associate a single project table with multiple database instances, which essentially creates duplicate tables.

You can use the MultiSource Option to connect to any relational data source that you access with an ODBC driver, including Microsoft Excel® files and text files. The MicroStrategy Engine uses multipass SQL to access multiple data sources and move data between them.

Support for duplicate tables

A duplicate table is a single project table that is mapped to more than one database instance. For example, in the scenario at the beginning of this chapter, the LU_REGION table exists in two data warehouses. You can bring both LU_REGION tables into the project. Although the tables are two separate physical tables in their respective data warehouses, they are treated as one logical table in the MicroStrategy project.

Your project can include lookup, relationship, and fact tables that are duplicated across multiple data sources. When you bring duplicate tables into the project, the following requirements must be satisfied:

- Duplicate tables must have identical table and column names.
- Columns that appear in duplicate tables must either have the same data type or compatible data types.

To maintain data consistency, the Engine applies data type compatibility rules when it joins columns in tables from different database instances.

- The number of columns in the table associated with the primary database instance must be less than or equal to the number of columns in the table associated with the secondary database instance. Any extra columns in the secondary table are not imported into the project.
- Duplicate tables should ideally contain an equal number of columns. If that is not the case, only the table associated with the secondary database instance can contain additional columns. Otherwise, you must treat the tables as two distinct tables.

SQL generation for MultiSource reports

If you have a report that uses objects from multiple data sources, the Engine performs two additional tasks to generate the SQL:

- It determines the optimal database instance to use for each pass of SQL.
- It identifies joins that need to occur across database instances.

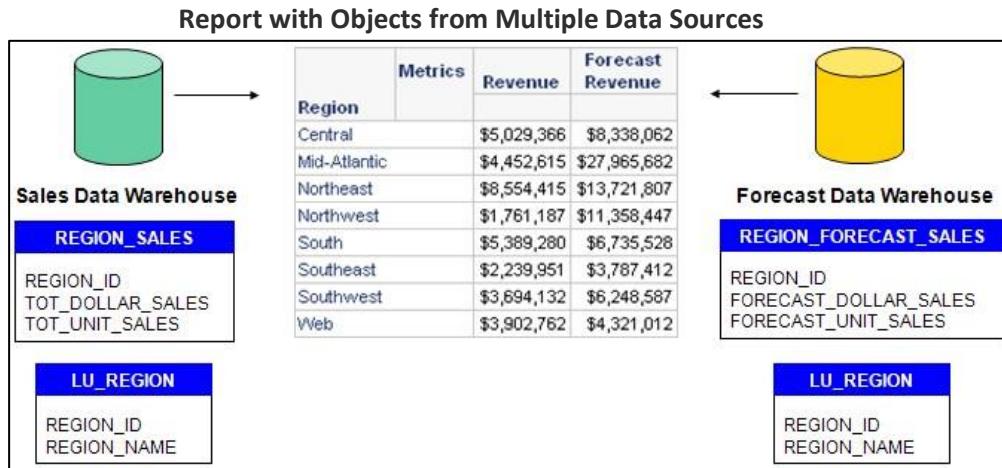
The MicroStrategy Analytics platform supports pre- and post-SQL statements for secondary databases in a multi-source report. You use the Pre/Post Statements VLDB setting to apply pre- and post-SQL statements. On a MultiSource report, you can execute multiple report pre- and post-SQL statements in a specified order for both primary and secondary databases. The SQL statements are also executed when the report is used in a document or dossier.

Designating primary and secondary tables

Every project table is initially mapped to a primary database instance. Duplicate tables can have primary and secondary database instances. The primary table exists in the primary database instance, while the secondary table is in the secondary database instance. You can update the primary database instance for a table, if necessary.

You can have multiple secondary tables if the table is mapped to more than one secondary database instance.

For example, consider the scenario described above:



In this example, each fact table maps to only one database. The primary database instance for the REGION_SALES table is the Sales Data Warehouse. The primary database instance for the FORECAST_SALES table is the Forecast Data Warehouse.

The LU_REGION table exists in both data warehouses, so you can map it to both database instances. You can assign either data warehouse as the primary or secondary database instance for this table. If you designate the Sales Data Warehouse as the primary database instance, the LU_REGION table from that database is the primary table. The LU_REGION table from the Forecast Data Warehouse is the secondary table.

Selecting the optimal data source for fact tables

When a table is available in multiple data sources, the Engine uses specific logic to select the optimal data source. SQL generation for reports is focused on metric data. When the Engine needs to calculate a metric, it identifies the best source for the underlying fact. To do this, the Engine weighs attributes in the template, the

metric's dimensionality, and report or metric filters. The Engine uses the following logic to select the optimal data source for a fact:

- The Engine calculates the metric using the primary database instance for the project if the fact comes from a fact table that is available in that primary database instance.
- If the fact comes from a fact table that is not available in the primary database instance for the project, the Engine calculates the metric using a secondary database instance. If the fact table is available in more than one secondary database instance, the Engine selects the database instance with the smallest GUID (alphabetically).

Selecting the optimal data source for lookup tables

After selecting the optimal data source for a fact, the Engine identifies the best source for corresponding attributes using the following logic:

- If the attribute comes from a lookup table that exists in the same data source as the one selected for the fact, the Engine obtains the attribute data from this same database instance.
- If the attribute comes from a lookup table that does not exist in the same data source as the one selected for the fact, the Engine obtains the attribute data from the primary database instance for the lookup table and moves it to the database instance used as the fact source.

This same logic also applies if the Engine has to retrieve attribute information from a relationship table. However, if a user is browsing attribute elements, the Engine treats lookup tables for attributes like fact tables. If the lookup table exists in the primary database instance for a project, the Engine queries that database instance. Otherwise, it uses the secondary database instance with the smallest GUID (alphabetically).

Joining data from multiple data sources

When the Engine needs to join data from multiple data sources, it selects data from the first data source in the memory of the Intelligence Server. Then it creates a temporary table in the second data source and inserts the data into this table to continue processing the result set.

If you have a data source that either does not support the creation of temporary tables or is not a suitable location for temporary tables, you can configure the database instance to be read-only. To do this, modify the database instance's CREATE and INSERT Support VLDB property to not support CREATE and INSERT statements. This setting prevents the Engine from creating tables in the data source.

To work with tables from multiple data sources, the Engine joins table columns based on the following list of data type compatibility rules:

Data Type	Compatible Data Type
BigDecimal	BigDecimal
Binary	Binary
CellFormatData	CellFormatData
Char	<ul style="list-style-type: none"> • Char • VarChar
Date	Date TimeStamp
Decimal	Decimal Integer with scale of 0 Numeric
Double	Double Float Real
Float	Double Float Real
Integer	Decimal with scale of 0 Integer Numeric with scale of 0
LongVarBin	LongVarBin
LongVarChar	LongVarChar

Data Type	Compatible Data Type
NChar	NChar
Numeric	Decimal Integer with scale of 0 Numeric
NVarChar	NVarChar

Real	Double Float Real
Time	Time TimeStamp
TimeStamp	Date Time TimeStamp
Unsigned	Unsigned
VarBin	VarBin
VarChar	<ul style="list-style-type: none"> • Char • VarChar

The Engine cannot join data from two columns that do not have compatible data types.

The data type of a column is based on the Engine data type definition; not the data type definition in the physical database. These two data type definitions may not always be the same.

In a given report's SQL, the Engine may identify a different data source for each pass. The Engine follows the data type compatibility rules and moves data between the data sources until the result set is complete.

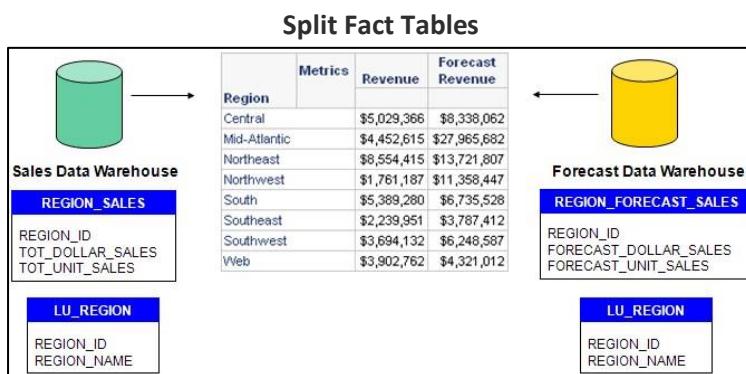
Common use cases

The following examples demonstrate scenarios where data must be retrieved data from multiple data sources. These examples are not intended to be a comprehensive list.

Split fact tables

In many organizations, multiple data sources are used to store different types of facts. As a result, your project may have individual reports that are dependent on multiple facts that each store their data in distinct data sources.

The following example shows split fact tables:

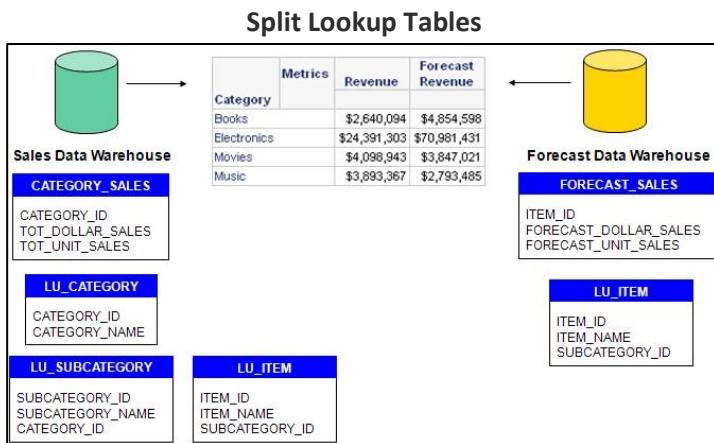


Users can obtain the region data from either data warehouse. However, revenue data is available only in the Sales Data Warehouse, and forecast revenue data is available only in the Forecast Data Warehouse.

Split lookup tables

Your organization may store lookup and relationship tables in a single data source, but it is also common for each table type to be stored in its own data source. If the latter is true, you may have a report that requires you to join data in one data source using relationships stored in another data source.

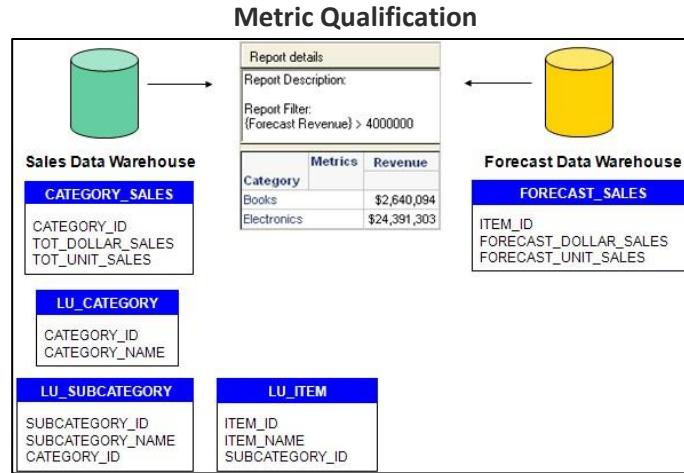
The following example shows split lookup tables:



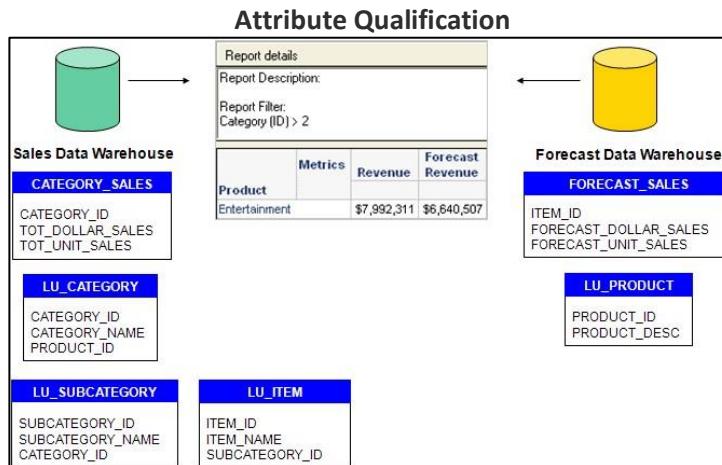
The lookup tables related to the Category, Subcategory, and Item attributes are stored in the Sales Data Warehouse. Forecast revenue is stored at the item level in the Forecast Data Warehouse. This data needs to be aggregated to the category level to calculate the forecast revenue for the report. Remember that the Forecast Data Warehouse stores only the relationship between Item and Subcategory. Therefore, this aggregation requires you to join the item-level forecast revenue data to the category data using the lookup tables in the Sales Data Warehouse.

Filtering qualifications

You may also have reports that filter data from one data source by qualifying on data that comes from another data source. The following image shows an example that involves a metric qualification:



The report contains a filter based on the Forecast Revenue metric. This fact data is stored in the Forecast Data Warehouse. The filter is used to qualify on the revenue for each category, which is stored in the Sales Data Warehouse. You can have this same scenario with other types of filter qualifications. The following example involves an attribute qualification:



The report contains a filter based on the Category attribute. This attribute is stored in the Sales Data Warehouse. However, the filter is used to determine the elements to include in the result set for the Product attribute, which is stored in the Forecast Data Warehouse.

These cases are just a few examples of how you can use the MultiSource Option to combine data from multiple sources in a single report. The MultiSource Option supports a variety of reporting needs, including:

- Supporting conformed dimensions across different data sources
- Accessing aggregate-level and detail-level data in different data sources
- Integrating operational data with your data warehouse

- Using distinct data sources for simple and complex queries
- Accessing multiple MicroStrategy BI implementations

The following exercise guides you through one possible application of the MultiSource Option.

Exercises

Exercise 9.1: Use the MicroStrategy MultiSource Option

The MultiSource option enables you to retrieve data from multiple sources. This allows analysts to create and view a single report that pulls data from different locations. In this exercise, you will create a report that retrieves data from one data source that stores actual sales information, and another data source that stores forecast figures.

You will create a report that displays the following attributes and metrics: Employee, Item, Revenue, and Forecast Revenue. The following are high-level steps for this exercise:

1. Set up the project to connect to tables in two distinct warehouses.
2. In the MicroStrategy Tutorial project, create a layer called MultiSource.
3. In the Project Tables View in Architect, add the following tables to the MultiSource layer: FORECAST_SALES, LU_EMPLOYEE, LU_ITEM, and ORDER_DETAIL.
4. In the FORECAST_SALES table, ensure that EMP_ID is mapped to the LU_EMPLOYEE table, and ITEM_ID is mapped to the LU_ITEM table.
5. Create the Forecast Revenue fact.
6. Save the MultiSource layer and update your schema.
7. Create the Forecast Revenue metric based on the Forecast Revenue fact.
8. Create a report with the Employee and Item attributes and the Revenue and Forecast Revenue metrics. When you execute the report, your result should resemble the following:

Employee	Item	Metrics	Revenue	Forecast Revenue
	100 Places to Go While Still Young at Heart		\$1,902	2,160
	Art As Experience		\$719	282
	The Painted Word		\$668	254
	Hirschfeld on Line		\$1,274	971
	Adirondack Style		\$1,293	979
	Architecture : Form, Space, & Order		\$1,338	1,417
	50 Favorite Rooms		\$683	617
	500 Best Vacation Home Plans		\$596	279
	Blue & White Living		\$730	335
	Ways of Seeing		\$748	231
	Gonzo, the Art		\$1,077	1,382
	Cabin Fever : Rustic Style Comes Home		\$503	391
	American Bungalow Style		\$1,410	1,558
	Building With Stone		\$462	730
	Voyaging Under Power		\$542	297
	Working With Emotional Intelligence		\$937	865
	Attention to Detail		\$745	529
	The 48 Laws of Power		\$523	761
	Don't Step in the Leadership		\$646	349

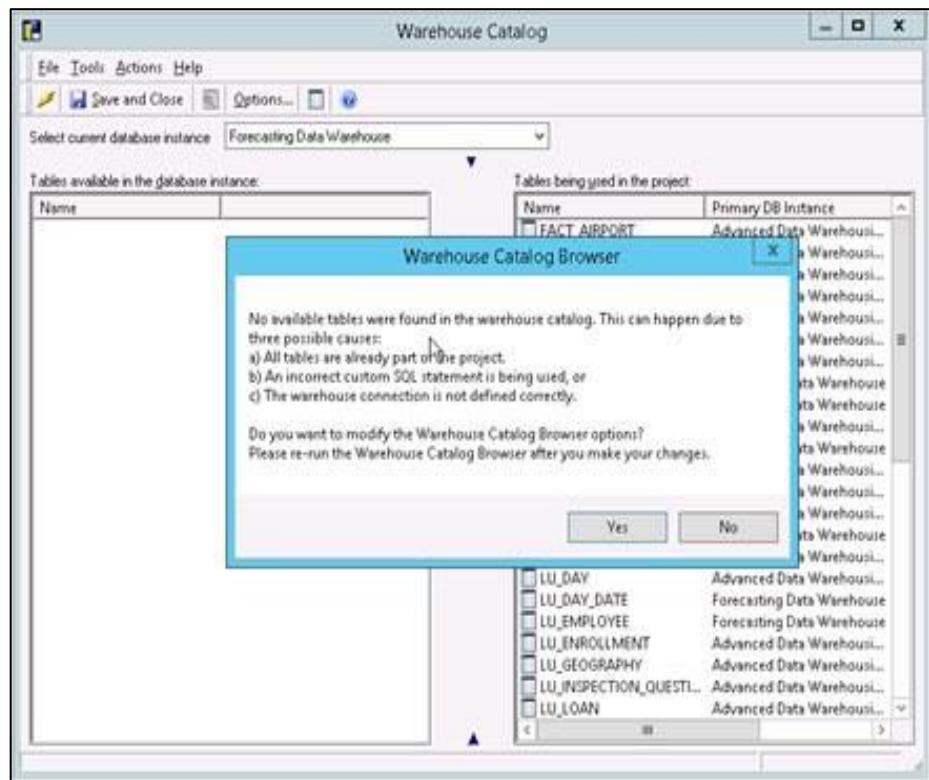
Add relevant tables from two different warehouses

1. In Developer, open the **MicroStrategy Tutorial Project**.

*If you were logged out, you can log in to **MicroStrategy Analytics Modules Project Source** using the following credentials:*

- login id: administrator
- password: (none)

2. From the **Schema** menu, select **Warehouse Catalog**.
3. In the Select current database instance drop-down box, select **Forecasting Data Warehouse**.
4. If the Warehouse Catalog Browser message window is displayed, click Yes to modify the **Warehouse Catalog Browser** options.

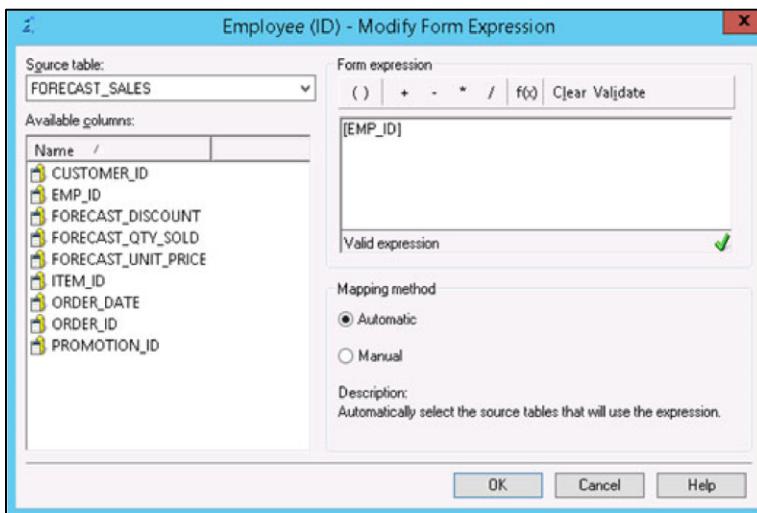


5. In the Warehouse Catalog window, click Read the Warehouse Catalog. The FORECAST_SALES table is displayed under Tables Available in the Database Instance pane.
6. Using the > arrow, move the FORECAST_SALES table to the Tables Being Used in the Project pane.
7. Click Save and Close.
8. Update the project schema.

Update the source tables for attributes

Update the Employee and Item attributes to use the FORECAST_SALES table as one of the source tables.

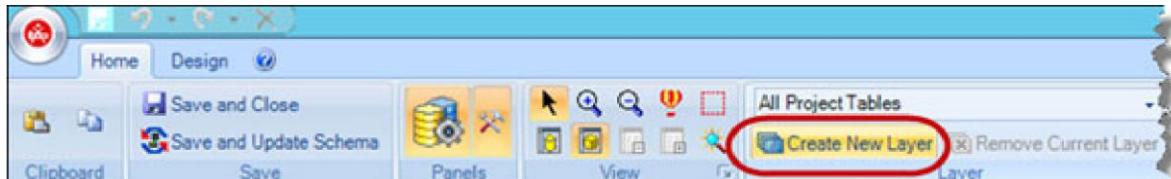
1. In the MicroStrategy Tutorial project, navigate to the **Schema Objects\ Attributes\Geography** folder and double-click the **Employee** attribute.
2. In the Attribute Editor, select the **ID** attribute form and click **Modify**.
3. In the Modify Attribute Form window, click **New**.
4. In the Modify Form Expression window, from the **Source Table** drop-down list, select **FORECAST_SALES**, and then double-click **EMP_ID** to add it to the Form Expression pane. Ensure that **Automatic** mapping is selected, and then click **OK**.



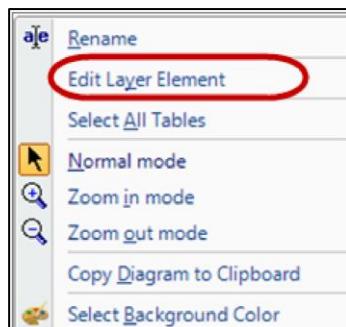
5. In the Modify Attribute Form window, in the Source tables pane, ensure that FORECAST_SALES is selected, and then click **OK**.
6. In the Attribute Editor, click **Save and Close**.
7. Repeat this set of steps to add the FORECAST_SALES table as one of the source tables for the Item attribute, with the following details:
 - The attribute is located in the Schema Objects\Attributes\Products folder.
 - In the Form Expression pane, use the ITEM_ID column from the FORECAST_SALES table.
 - After you create the new expression, In the Attribute Editor message window, click **Leave All**.
8. Update the project schema.

Create a layer that includes the relevant tables

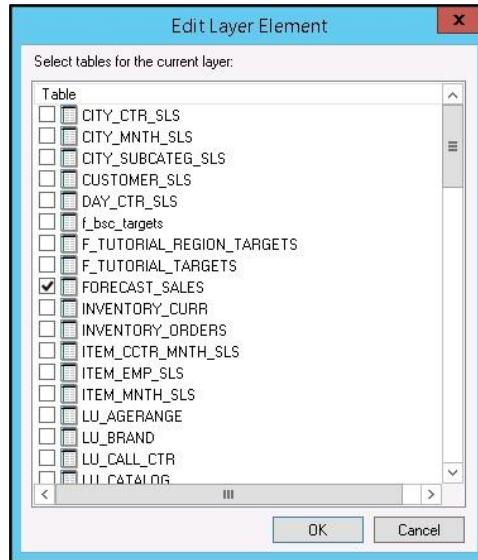
1. From the **Schema** menu, select **Architect**. If the Read Only window is displayed, select **Edit** and click **OK**.
2. In Architect, click the **Project Tables View** tab.
3. On the toolbar, in the Layer area, click **Create New Layer**.



4. In the MicroStrategy Architect window, type **MultiSource** as the name of your layer.
5. Click **OK**.
6. In the Project Tables View tab, right-click in an empty area and select **Edit Layer Element**.



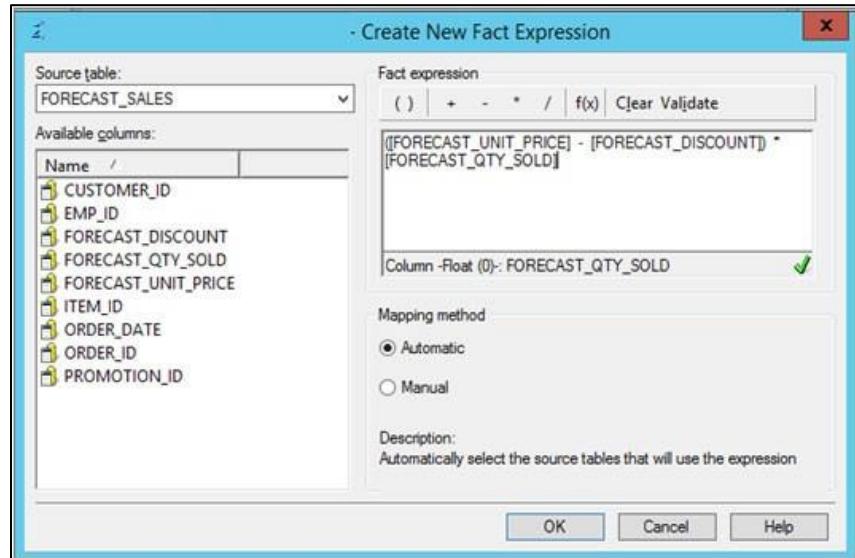
7. In the Edit Layer Element window, select the following tables:
 - FORECAST_SALES
 - LU_EMPLOYEE
 - LU_ITEM
 - ORDER_DETAIL



8. Click **OK** to add the selected tables to the MultiSource layer.
-

Create the Forecast Revenue fact

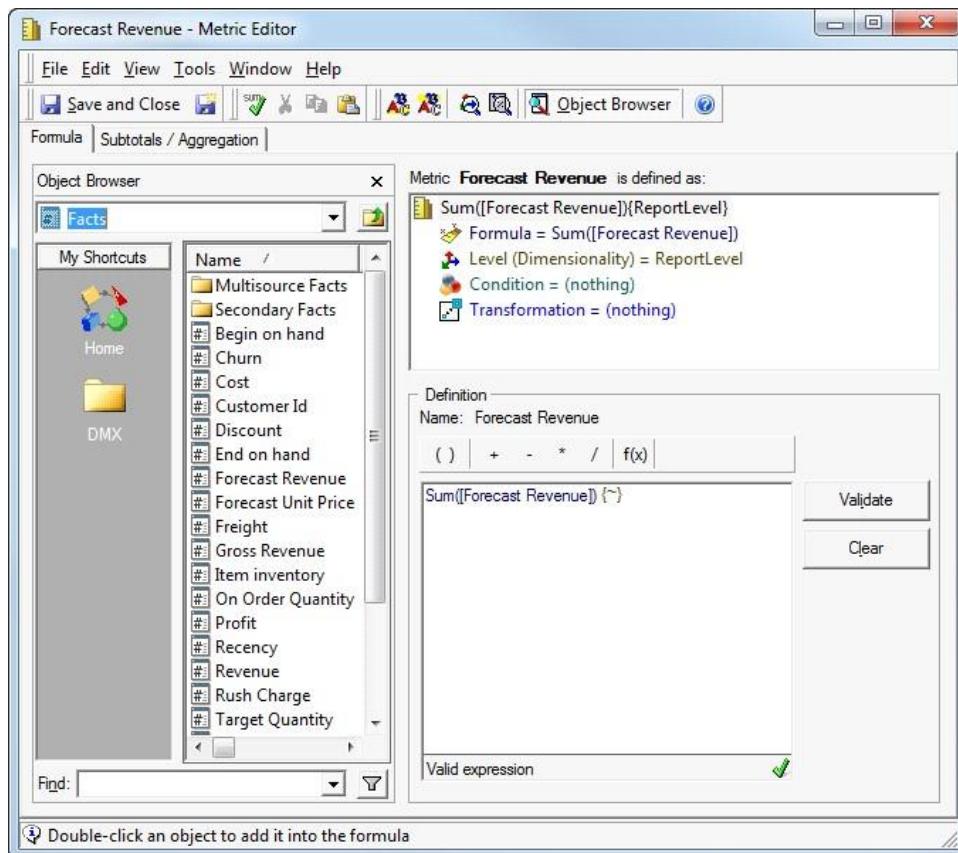
1. Right-click the **FORECAST_SALES** table header and select **Create Fact**.
2. In the Create New Fact Expression window, from the Source Table drop-down list, select **FORECAST_SALES**.
3. Create the following expression:
$$(\text{FORECAST_UNIT_PRICE} - \text{FORECAST_DISCOUNT}) * \text{FORECAST_QTY_SOLD}$$
4. Ensure **Automatic mapping** is selected.



5. Click **OK**.
6. In the FORECAST_SALES table, right-click the fact column that you created and select **Rename**.
7. In the MicroStrategy Architect window, type **Forecast Revenue**.
8. Click **OK**. Then click **Save and Close**.

Create the Forecast Revenue metric

1. In Developer, in the MicroStrategy Tutorial project, in the Public Objects\ Metrics\Sales Metrics folder, from the **File** menu, point to **New**, and select **Metric**.
2. In the New Metric window, keep **Empty Metric** selected and click **OK**.
3. In the Metric Editor, double-click the **Forecast Revenue** fact to define the new metric as **Sum([Forecast Revenue])**.



4. Click **Save and Close**.
 5. Save the metric as **Forecast Revenue**.
-

Create the MultiSource Revenue and Forecast Revenue report:

1. In the MicroStrategy Tutorial project, in the Public Objects\Reports folder, create a new report with
 - The Employee and Item attributes on the rows
 - The Revenue and Forecast Revenue metrics on the columns
2. Execute the report. The report displays Revenue data from the Tutorial warehouse and Forecast Revenue data from the Forecast Data warehouse:

Employee	Item	Metrics		Forecast	
		Revenue		Revenue	
	100 Places to Go While Still Young at Heart	\$1,902		2,160	
	Art As Experience	\$719		282	
	The Painted Word	\$668		254	
	Hirschfeld on Line	\$1,274		971	
	Adirondack Style	\$1,293		979	
	Architecture : Form, Space, & Order	\$1,338		1,417	
	50 Favorite Rooms	\$683		617	
	500 Best Vacation Home Plans	\$596		279	
	Blue & White Living	\$730		335	
	Ways of Seeing	\$748		231	
	Gonzo, the Art	\$1,077		1,382	
	Cabin Fever : Rustic Style Comes Home	\$503		391	
	American Bungalow Style	\$1,410		1,558	
	Building With Stone	\$462		730	
	Voyaging Under Power	\$542		297	
	Working With Emotional Intelligence	\$937		865	
	Attention to Detail	\$745		529	
	The 48 Laws of Power	\$523		761	
	Don't Step in the Leadership	\$646		349	

3. Save the report in the Public Objects\Reports folder as **MultiSource Revenue and Forecast Revenue**.

CHAPTER 10: FACT EXTENSIONS AND DEGRADATIONS

So far you have learned to use basic schema design principles to develop your MicroStrategy architecture. Next, you will learn to leverage advanced design feature to refine your business intelligence application and satisfy complex requirements.

In this lesson, you will learn to use advanced design features in the MicroStrategy platform to configure fact level extensions, partition tables, create logical views, address attribute roles, and develop MTDI cubes.

Fact level extension

Fact level is determined by the attribute IDs present in a fact table. Analysts can report on a fact at the level at which it is stored in the data warehouse, or they can aggregate the fact to a higher level of detail. However, facts cannot be aggregated to a lower level of detail that does not exist in the data warehouse.

For example, the UNIT_SALES fact below can be aggregated at a week level or higher. But it cannot be reported at the daily level because that level of granularity does not exist in the table.

FACT_UNIT_SALES	
Fact Level	{
ITEM_ID	
WEEK_ID	
UNIT_SALES	

It is possible for the same fact to be stored at multiple attribute levels within a hierarchy. For example, your warehouse may contain another fact table that stores unit sales by item and date, rather than item and week. This fact table stores unit sales for items at a lower level within the Time hierarchy.

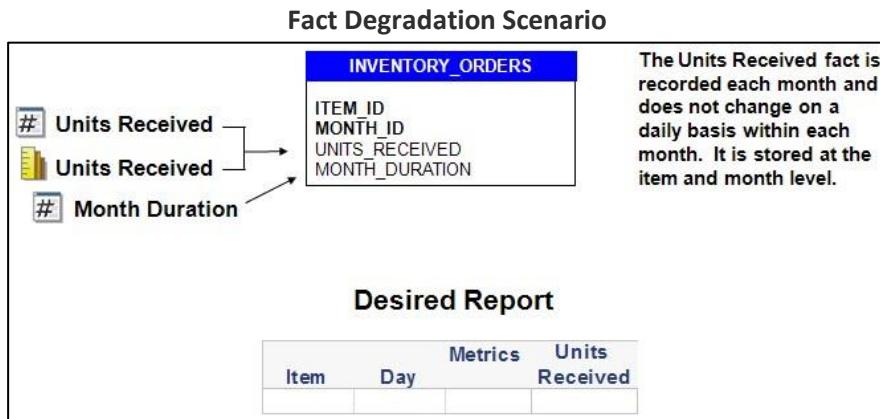
Architect provides the following types of fact level extensions:

- **Degradation:** Lowers the fact to a level within the same hierarchy.
- **Extension:** Extends the fact to include a level from a different hierarchy that is not currently related to the fact.
- **Disallow:** Restricts a fact level to prevent unnecessary cross joins with lookup tables.

Fact degradation

A fact degradation lowers a fact to a level that exists within the same hierarchy.

In the following image, fact data is stored at the month level. An analyst who is unaware of the Units Received fact level creates a report that aggregates at the Day level:



In this example, the report contains the Units Received metric that uses the Units Received fact in its definition. If the above report is executed, the requested data will not be returned because the Units Received fact is stored at the Item and Month levels. A request for Units Received fact data at the Day level returns the following error:



The error message notifies users that the Units Received fact is not available at the Day and Item levels. To return data at the desired level, you must use a fact degradation to lower the level of the Units Received fact from Month to Day.

Alternatively, you can change the level of the fact table in the data warehouse itself. However, changing the fact table is not always an option. The fact data may not be captured at the desired level in the source system, or there may be other organizational or environmental restrictions that prevent changes to the table structure.

Steps for fact degradation

Implement the following steps to create a fact degradation:

Selecting the target attribute

To create a fact degradation, you must first select a target attribute that is in the same hierarchy as another attribute in the fact table. The target attribute must exist at a lower level in the hierarchy. For example, Units Received is in a fact table that contains the Month_ID column. The Time hierarchy contains the Month attribute at a level above the Day attribute. Therefore, the Day attribute is a valid target for the Units Received fact degradation.

Selecting the join attribute

A join attribute is used by the SQL Engine to join the fact table to the selected target attribute. The join attribute must be at a level above the target attribute in the same hierarchy.

For example, for the Units Received fact degradation, the Units Received fact is stored at the item and month level, so it is related to both the Item and Month attributes. Since the fact must be lowered to the day level, select Month as the join attribute because it is directly related to the Day attribute. The Item attribute is not directly related to the Day attribute, so it cannot be used as the join attribute.

Determining the join direction

The join direction indicates how the fact joins to the target attribute. The join can occur using only the join attribute, or it can also join to children of the join attribute.

For example, for the Units Received fact degradation, the Units Received fact is stored only at the item and month level. It is not stored at any other levels of time. You could join only against the attribute itself (Month). However, if the Units Received fact were stored at another level of time between month and day, such as week, you could join to the fact at the month or week level.

If you allow the SQL Engine to join against the join attribute and any of its children, ensure that the allocation expression you use for the fact degradation returns values that are valid at all of those attribute levels.

Defining an allocation expression

An allocation expression calculates the fact values at the desired degradation level. An allocation expression can include attributes, facts, constants, and any standard expressions, including mathematical operators, pass-through functions, and so on.

For example, the Units Received fact is stored in the data warehouse at the month level. To derive the value at the day level, an allocation expression must be used to translate the monthly values. For the Units Received degradation, you would create an allocation expression to divide the monthly units received by the duration of the month: ($[Units Received] / [Month Duration]$). This will yield a rough approximation of units received at the day level.

Exercises

Exercise 10.1: Create a fact level degradation

The high-level steps for this exercise are:

1. Create the following report in the Forecasting Project:

Report View: 'Local		Switch to:
	Metrics	Forecast Cost
Quarter		

2. Run the report. Add the Month attribute to the template and run the report again. After reviewing the error message, save the report as Degradation Example in the My Personal Objects\My Reports folder.
3. Next, create a degradation for the Forecast Cost fact to enable reporting on that fact at the Month level. In the data warehouse, this fact exists only at the Quarter level. Use the following allocation expression for the fact degradation:
[Forecast Cost] / 3. After creating the fact degradation, update the project schema.
4. Run the report. The result set should look like the following:

Quarter	Month	Metrics	Forecast Cost
2014 Q1	Jan 2014		1,034,178
	Feb 2014		1,034,178
	Mar 2014		1,034,178
2014 Q2	Apr 2014		1,180,179
	May 2014		1,180,179
	Jun 2014		1,180,179
2014 Q3	Jul 2014		1,364,618
	Aug 2014		1,364,618
	Sep 2014		1,364,618
2014 Q4	Oct 2014		1,539,513
	Nov 2014		1,539,513
	Dec 2014		1,539,513
2015 Q1	Jan 2015		1,385,414
	Feb 2015		1,385,414
	Mar 2015		1,385,414
2015 Q2	Apr 2015		1,392,951
	May 2015		1,392,951
	Jun 2015		1,392,951
Jul 2015			1,577,509

The image above only displays the first few rows of the result set for the report.

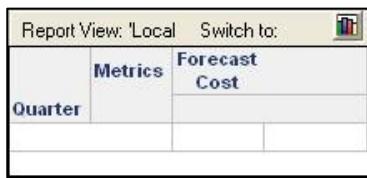
Create a report

1. In Developer, open the **Forecasting Project**.

If you were logged out, you can log in to **MicroStrategy Analytics Modules Project Source** using the following credentials:

- login id: administrator
- password: (none)

2. Create the following report in the **My Personal Objects\My Reports** folder:



Report View: 'Local' Switch to: 		
	Metrics	Forecast Cost
Quarter		

- You can access the **Quarter** attribute in the **Schema Objects\Attributes\ Forecasting** folder.
- The **Forecast Cost** metric can be found in the **Public Objects\Metrics\ Forecasting** folder.

3. Run the report. The result set should resemble the following:



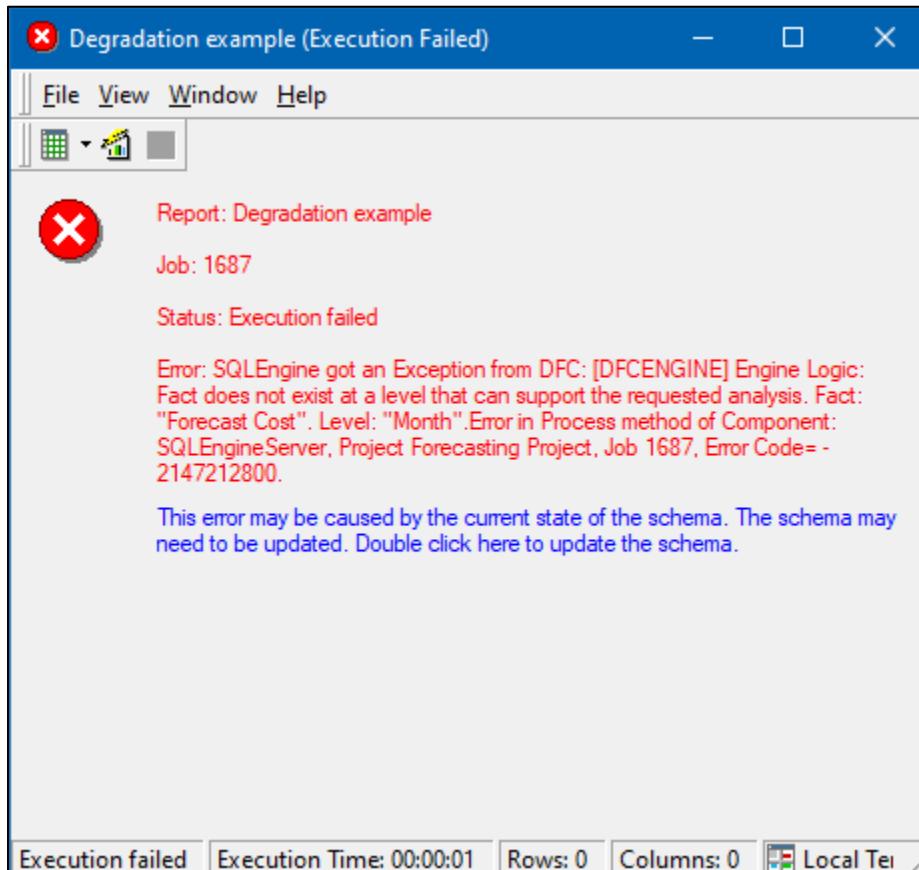
Quarter	Metrics	Forecast Cost
2011 Q1		\$3,102,533
2011 Q2		\$3,540,538
2011 Q3		\$4,093,854
2011 Q4		\$4,618,538
2012 Q1		\$4,156,241
2012 Q2		\$4,178,852
2012 Q3		\$4,732,527
2012 Q4		\$5,525,563
2013 Q1		\$5,052,528
2013 Q2		\$5,552,141
2013 Q3		\$5,791,716
2013 Q4		\$6,463,698

The Forecast Cost metric is aggregated at the quarter level.

Modify the report to include the Month attribute

4. Switch to **Design View**.

5. Add the **Month** attribute to the report template to the right of Quarter.
 - The **Month** attribute is located in the **Schema Objects\Attributes\ Forecasting** folder.
6. Run the report. The following error message is displayed:



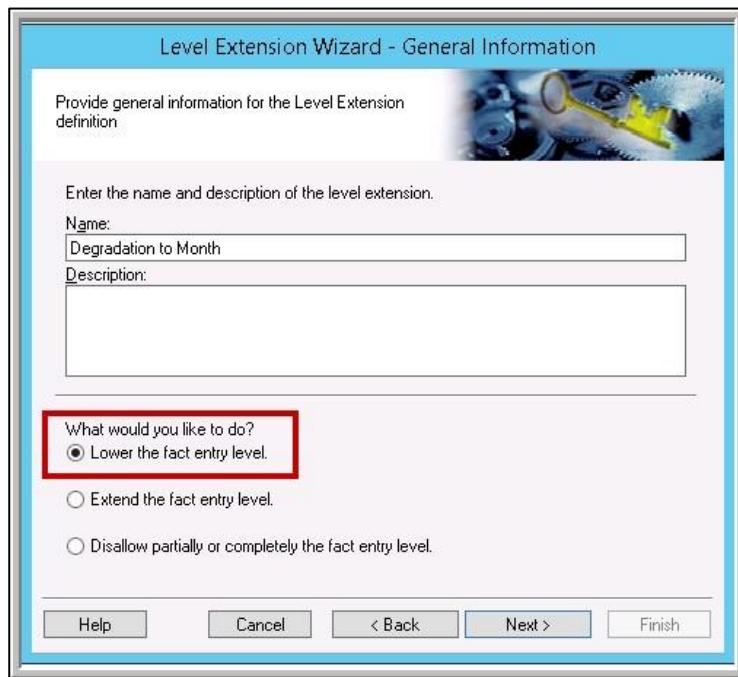
The error occurs because the Forecast Cost fact does not exist at the month level in the data warehouse. To report on that fact at a level lower than quarter, you must create a fact degradation.

Save the report

1. Switch to **Design View**.
2. Save the report in the My Personal Objects\My Reports folder as **Degradation Example**.
3. Close the report.

Create a fact degradation at the month level for the Forecast Cost fact

1. In the Schema Objects\Facts\Forecasting folder, open the **Forecast Cost** fact in the Fact Editor.
2. In the Fact Editor, click the **Extensions** tab.
3. Click **New**.
4. In the Level Extension Wizard, in the Introduction window, click **Next**.
5. In the General Information window, in the **Name** box, type **Degradation to Month** as the name for the extension.
6. Under What Would You Like To Do?, click **Lower the fact entry level**.



7. Click **Next**.
8. In the Extended Attributes window, select the **Show all attributes** check box.
9. From the **Available Attributes** pane, select the first **Month** attribute and click the **>** button to add it to the **Selected Attributes** list.

Multiple attributes can be added for a single fact degradation. However, you must ensure that the allocation expression returns correct results for all attribute levels.

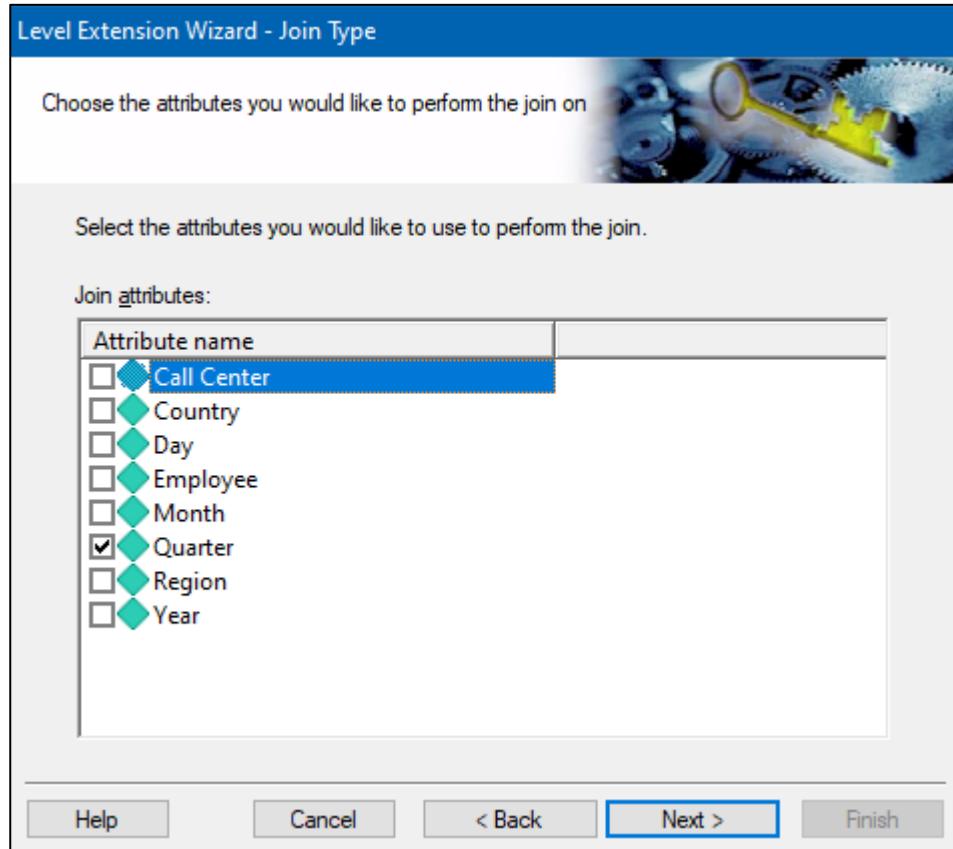
For example, if a Day attribute is added to the degradation definition, create a single allocation expression that returns the correct results at both the month and day levels.

- The Month attribute is located in the Schema Objects\Attributes\ Forecasting folder.

10. Click **Next**.

11. In the Join Type window, select the first **Quarter** attribute check box.

- The Quarter attribute is located in the Schema Objects\Attributes\ Forecasting folder.



12. Click **Next**.

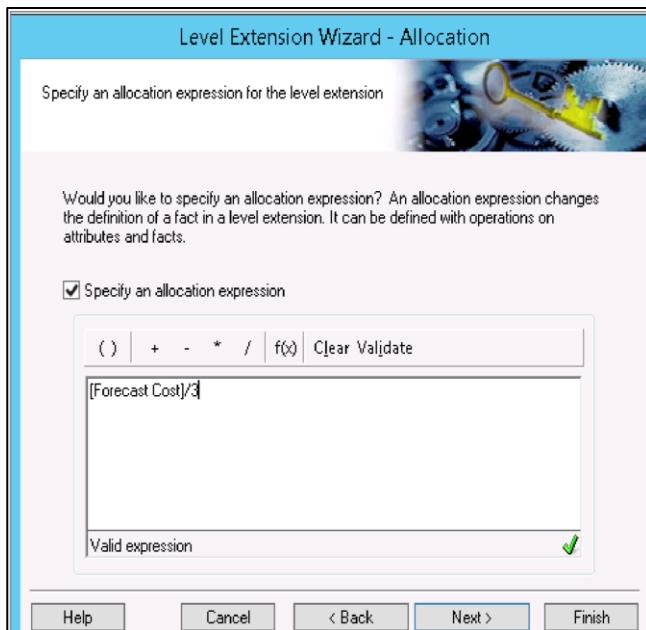
13. In the Join Attributes Direction window, in the Join Attributes pane, in the **Join against column**, keep the default setting.

14. Click **Next**.

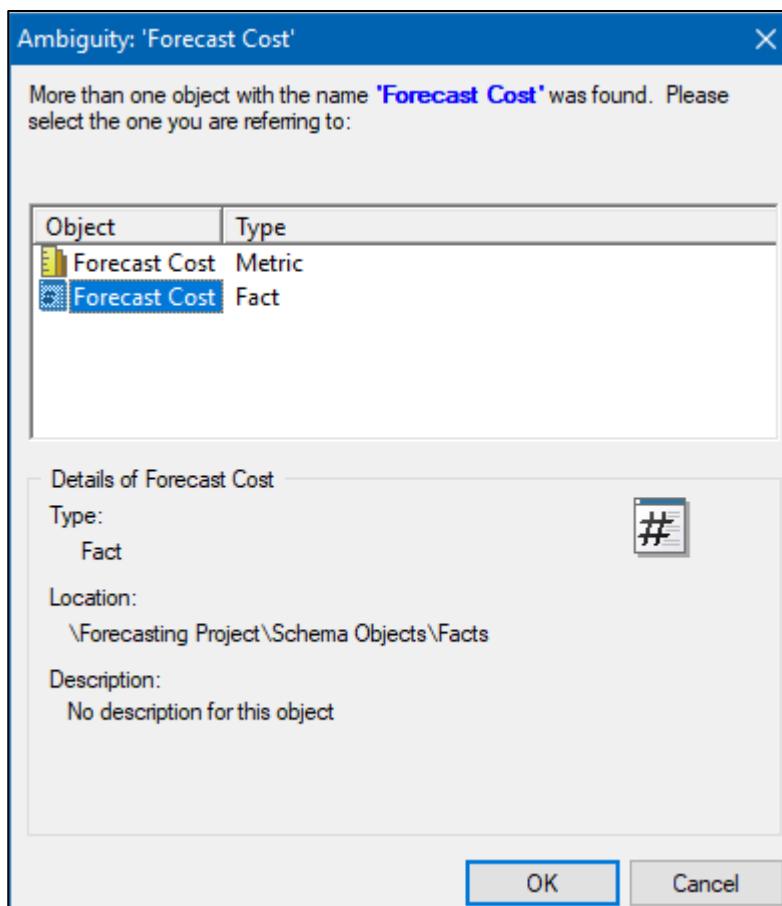
15. In the Allocation window, select the **Specify an allocation expression** check box.

16. In the box, type the following expression:

[Forecast Cost] / 3



17. Click **Validate** to validate your expression.
18. If the Ambiguity: Forecast Cost window is displayed, select the **Forecast Cost** fact and click **OK**.



19. Click **Next**. In the Finish window, review the information and click **Finish**.

*If necessary, click **Back** to go back through the Level Extension Wizard and make any changes.*

20. In the Fact Editor, click **Save and Close**. Then update the project schema.

Run the report

1. Run the Degradation Example report. The result set should resemble the following:

Quarter	Month	Metrics	Forecast Cost
2011 Q1	Jan 2011	\$1,034,178	
	Feb 2011	\$1,034,178	
	Mar 2011	\$1,034,178	
2011 Q2	Apr 2011	\$1,180,179	
	May 2011	\$1,180,179	
	Jun 2011	\$1,180,179	
2011 Q3	Jul 2011	\$1,364,618	
	Aug 2011	\$1,364,618	
	Sep 2011	\$1,364,618	
2011 Q4	Oct 2011	\$1,539,513	
	Nov 2011	\$1,539,513	
	Dec 2011	\$1,539,513	
2012 Q1	Jan 2012	\$1,385,414	
	Feb 2012	\$1,385,414	
	Mar 2012	\$1,385,414	



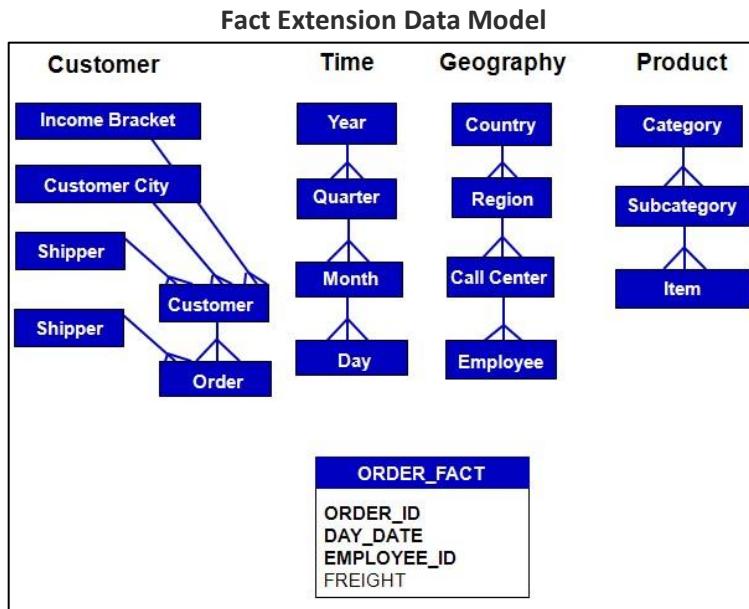
You can now report on the Forecast Cost fact at the month level. The monthly values are estimates based on the allocation expression you provided in the definition of the fact degradation. Notice that the forecast cost value is the same for each month in a given quarter.

2. Close the report.

Extending fact levels

A fact extension extends a fact to a level in an unrelated hierarchy. You can use fact extensions when analysts want to report on fact data based on an unrelated attribute level that does not exist in the data warehouse.

A fact level extension is outlined in the following example. Consider the following simplified data model for the MicroStrategy Tutorial project:



In this data model, the Freight fact is stored in the data warehouse at the Employee, Order, and Day levels. Analysts now want to create reports to analyze freight data at the Item level, but the Freight fact is not related to any attribute in the Product hierarchy.

When analysts create a report that includes the Item attribute and a Freight metric, the result set resembles the following:

Item and Freight—Report Result Set		
Item	Metrics	Freight
100 Places to Go While Still Young at Heart		\$ 1,234,199
Art As Experience		\$ 1,234,199
The Painted Word		\$ 1,234,199
Hirschfeld on Line		\$ 1,234,199
Adirondack Style		\$ 1,234,199
Architecture : Form, Space, & Order		\$ 1,234,199
50 Favorite Rooms		\$ 1,234,199
500 Best Vacation Home Plans		\$ 1,234,199
Blue & White Living		\$ 1,234,199
Ways of Seeing		\$ 1,234,199
Gonzo, the Art		\$ 1,234,199

The image above displays only the first few rows of the report.

The report returns the same freight value for each item. This result set is meaningless because the freight fact is not stored at the Item level. Because no relationship exists between Item and Freight, the SQL Engine performs a cross join between the fact and lookup tables to retrieve the data for the report. The following image shows the SQL for this report:

Item and Freight—Report SQL

```
select    a12.ITEM_ID ITEM_ID,
          max(a12.ITEM_NAME) ITEM_NAME,
          sum(a11.FREIGHT) WUXBFS1
  from      `order_fact`      a11
            cross join `lu_item` a12
 group by  a12.ITEM_ID
```

To view freight information by Item, you have to extend the Freight fact to the Item level. You cannot use a fact degradation because Item is an attribute from a hierarchy that does not have a relationship with the attributes related to the Freight fact.

As an alternative to creating a fact extension, you can add a new level to the fact table in the data warehouse itself. However, changing the fact table is not always an option. The fact data may not be captured at the desired level in the source system, or there may be organizational or environmental restrictions that prevent you from changing the table structure.

You can use the following methods to create fact extensions, based on the desired joining behavior between the target attribute and fact:

- **Table relation:** Select a specific table to use for the join. Select this option if you always want the SQL Engine to use the same table to join the desired attribute and fact.
- **Fact relation:** Instead of selecting a single table, you select a fact to use for the join. This option enables the SQL Engine to use any table containing the fact to join the desired attribute and fact. Select this option if you want to allow the SQL Engine to choose the optimal table for a particular query.
- **Cross product:** Enable the SQL Engine to perform a cross join between the lookup table of the desired attribute and the fact table.

Use this option if there are no tables in the data warehouse that you can use to join the desired attribute and fact. A cross-join requires significant processing overhead, and the resulting data may not be meaningful.

Fact extension using the table relation method

The table relation method for a fact extension forces the SQL Engine to always join the desired attribute and fact using a specified table. Implement the following steps to create a fact extension using a table relation.

Identify the target attribute level

Analysts will notify you when a fact does not aggregate to the desired attribute level on a report. When this happens, identify the hierarchy that includes the desired attribute. If you want to report on the fact at all levels in the hierarchy, select the lowest-level attribute in that hierarchy.

For example, for the Freight fact extension, if you want to report on the Freight fact at any attribute level in the Product hierarchy, select the Item attribute, which is the lowest-level attribute in the hierarchy. Extending the Freight fact to the Item level enables users to create reports that analyze freight data using any attribute from the Product hierarchy. If you select a higher-level attribute from the Product hierarchy, such as Subcategory, the fact extends only to that attribute level or above.

Selecting the table for the join

The SQL Engine needs to join the table that contains the extending fact to the lookup table that stores the target attribute. Because these two tables are not related, there is no need to select another data warehouse table to serve as a relationship table between the fact and lookup tables.

When you specify the target attribute, Architect searches the project warehouse catalog and returns a list of tables that contain the ID column of that attribute. Scan the list of candidate tables to identify the optimal table for the join. As you examine each table in the list, consider the number of possible join paths, the optimal join path for a given allocation expression, and any other characteristics specific to your data warehouse environment. For example, you may choose a table that includes reliable indexes, or one that is frequently updated.

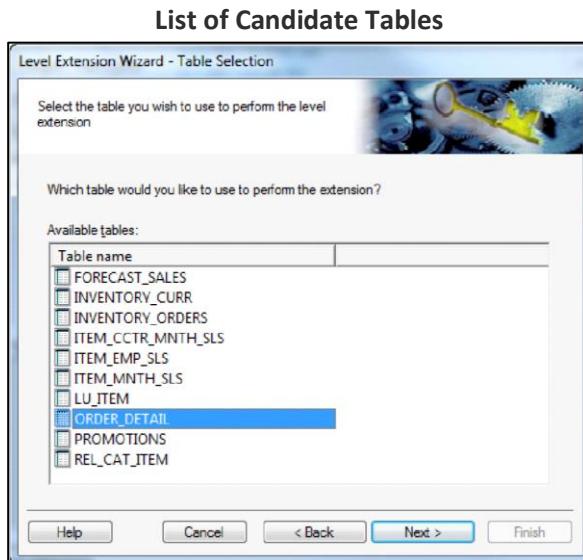
In the freight reporting example, the Freight fact is stored in the ORDER_FACT table. The following image shows the logical view for the ORDER_FACT table:

Objects present in table 'ORDER_FACT':	
Name	Type
Employee	Attribute
Day	Attribute
Days to Ship	Attribute
Customer	Attribute
Order	Attribute
Payment Method	Attribute
Ship Date	Attribute
Shipper	Attribute
Phone Usage	Attribute
Freight	Fact

Notice that several attributes from multiple hierarchies map to the ORDER_FACT table. All of these attributes relate to the Freight fact and are possible candidates to relate the Freight fact to the Item attribute.

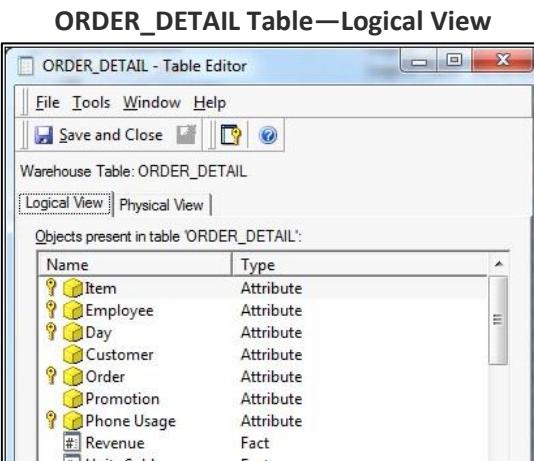
The ORDER_FACT table is the only table in the data warehouse that contains the Freight fact. Therefore, you have to join the ORDER_FACT table to the LU_ITEM table to relate the Freight fact to the Item attribute.

When the Freight fact is extended to Item using a table relation, Architect returns the following list of candidate tables:



The list of candidate tables includes the LU_ITEM and REL_CAT_ITEM tables. These lookup and relationship tables contain only product-related information. You can eliminate these tables as join candidates because no attributes from the Product hierarchy are related to the Freight fact. In general, you can usually eliminate tables from the target attribute's hierarchy because these tables typically do not provide a join path to other hierarchies.

The remaining tables are all fact tables that contain the Item attribute. Most of these tables have only one or two attributes in common with the ORDER_FACT table. The ORDER_DETAIL table, however, contains many of the same attributes as the ORDER_FACT table, including Employee, Day, Customer, and Order. The following image shows the logical view for the ORDER_DETAIL table:



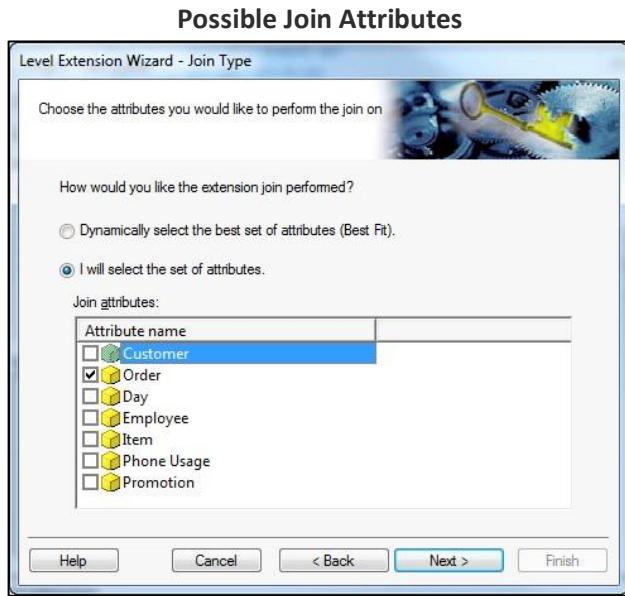
Use the ORDER_DETAIL table to join the LU_ITEM and ORDER_FACT tables using any of the common attribute columns. Because the ORDER_DETAIL table provides multiple join paths, it is the best table to use to join the Freight fact to the Item attribute.

The ORDER_DETAIL table is the optimal join table, provided you can use it in conjunction with the allocation expression for the fact extension. Examine the table's environmental characteristics to determine whether it is a suitable candidate. For example, if this table is only updated monthly and you want reports that provide current data, it would not be an ideal table to use for the join.

Selecting the join attribute or set of join attributes

After you select a table for the join, select an attribute or set of attributes from that table to target for the join. Architect provides a list of attributes whose ID columns are present in the join table. You can manually select the join attributes, or allow the SQL Engine to select the join attributes dynamically on a query-by-query basis.

For the Freight fact extension, select Order as the join attribute. The list of candidate join attributes is displayed in the Level Extension Wizard, as shown below:



Because the allocation expression for this fact extension uses facts that are related to individual orders, Order is the optimal join attribute.

Determining the join direction

There are two possible join directions. You can allow the join to only occur on the target attribute, or on the target attribute and its children. If the SQL Engine is allowed to dynamically select the join attributes, you cannot select the join direction. However, if you manually selected the join attributes, you must specify how the SQL Engine performs the join between the join attributes and the fact.

If the SQL Engine is allowed to join against the join attributes and any of their children, ensure that the allocation expression used for the fact extension returns valid values for any of those attribute levels.

For the Freight fact extension, allow the SQL Engine to join only against the Order attribute itself. Because the Order attribute is already the lowest-level attribute in the Customers hierarchy, it does not have any child attributes that can be used to join to the fact.

Defining an allocation expression

An allocation expression calculates fact values at the desired level. A valid allocation expression can include attributes, facts, constants, and any standard expression syntax, including mathematical operators, pass-through functions, and so on.

For example, for the Freight fact extension, you might create the following allocation expression:

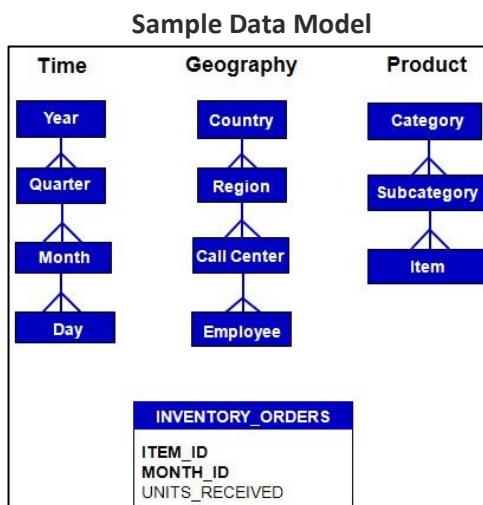
$(\text{Freight} * [\text{Item-level Units Sold}]) / [\text{Order-level Units Sold}]$

In the allocation expression, the value of the Freight fact at the Order level is proportionally distributed among items sold in this particular order. If there are 3 units of the same item in an order of 10 items, and the freight for this order was \$100, then the item-level freight for that particular item is \$30.

Disallowing fact levels

A fact disallow prevents a fact from being reported at a specific level. This function avoids unnecessary cross joins between fact and lookup tables. You can disallow a fact level if the resource cost for performing the operation is too expensive.

For example, a project may contain the following hierarchies and fact table:



This project contains Geography, Product, and Time hierarchies. The INVENTORY_ORDERS fact table stores data only at the level of Item and Month, so it is only related to the Product and Time hierarchies.

What if a user wants to run a report that displays a Units Received metric (built from the Units Received fact) as well as the Item and Call Center attributes? The Item attribute is related to the Units Received fact because it is part of the Product hierarchy, but the Call Center attribute is not related to this fact. The report would display the following result set:

Report Result Set—Unrelated Attribute and Fact

Item	Call Center	Metrics	Units Received
	Atlanta		1,470
	San Diego		1,470
	Berlin		1,470
	San Francisco		1,470
	Washington, DC		1,470
	Salt Lake City		1,470
	Miami		1,470
	Milwaukee		1,470
100 Places to Go While Still Young at Heart	New Orleans		1,470
	Seattle		1,470
	Boston		1,470
	New York		1,470
	London		1,470
	Fargo		1,470
	Memphis		1,470
	Paris		1,470
	Charleston		1,470
	Web		1,470
	Atlanta		2,135
	San Diego		2,135
	Berlin		2,135
	San Francisco		2,135
	Washington, DC		2,135
	Salt Lake City		2,135
	Miami		2,135

Because there is no way **to** relate call center data to units received data, this report displays the same number of units received for every call center associated with each item.

To retrieve data for the report, the SQL Engine generates SQL that results in a cross join between the lookup table for the Call Center attribute and the fact table that contains the units received data. The SQL for this report resembles the following:

Report SQL—Unrelated Attribute and Fact

```

select    a11.ITEM_ID ITEM_ID,
          max(a13.ITEM_NAME) ITEM_NAME,
          a12.CALL_CTR_ID CALL_CTR_ID,
          max(a12.CENTER_NAME) CENTER_NAME,
          sum(a11.UNITS_RECEIVED) 'WJXBFS1'
from      `inventory_orders` a11
          cross join `lu_call_ctr`      a12
          join     `lu_item`            a13
          on       (a11.ITEM_ID = a13.ITEM_ID)
group by  a11.ITEM_ID,
          a12.CALL_CTR_ID
  
```

The Units Received Fact is not related to any attribute in the Geography hierarchy, so the SQL Engine performs a cross join to the Call Center lookup table.

The report SQL includes the LU_CALL_CTR table in the FROM clause, but it cannot join this table to the INVENTORY_ORDERS fact table in the WHERE clause. The cross join makes it possible for the report to return data, but it can only produce a cross product, which does not yield a meaningful result set.

If you determine that there is no need for this cross join or the result set it produces, you can disallow the Call Center attribute for the Units Received fact. The fact disallow prevents the SQL Engine from executing the cross join.

Fact disallows prevent cross joins, but do not prevent normal joins from occurring. For example, for the fact table in the example above, disallowing an attribute from the Time or Product hierarchies for the Units Received fact would not prevent users from running a report with attributes from either hierarchy. The Units Received fact is related to attributes from each of these hierarchies, so a cross join to lookup tables would not occur.

CHAPTER 11: PARTITIONING

Partitioning is the division of a larger table into smaller tables. Partitioning is often implemented in a data warehouse to improve query performance by reducing the number of records that queries must scan to retrieve a result set. Partitioning is also used to decrease the amount of time necessary to load data into warehouse tables and perform batch processing.

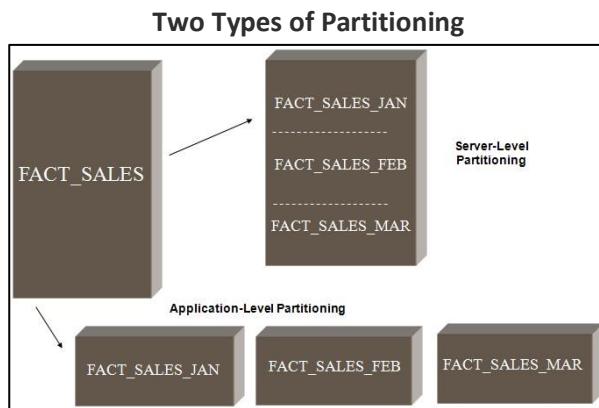
Databases differ dramatically in the size of the data files and physical tables they can manage effectively. Partitioning support varies by database. Most database vendors provide some support for partitioning at the database level. The use of a partitioning strategy is essential to designing a manageable data warehouse. Like all data warehouse tuning techniques, you should periodically re-evaluate your partitioning strategy.

Partitioning fact tables in the MicroStrategy schema

There are two basic types of partitioning:

- Server level
- Application level

The following diagram shows the difference between these two types of partitioning strategies:



Server-level partitioning divides one physical table into logical partitions in the database environment. The database software handles this type of partitioning completely, so these partitions are effectively transparent to MicroStrategy. Because only one physical table exists, the SQL Engine only writes SQL against a single table, and the database manages the logical partitions used to resolve the query.

Application-level partitioning divides one large table into several separate, smaller physical tables called partition base tables (PBTs). You split the table into smaller tables in the database itself, and then MicroStrategy manages the partitions used for a given query. Because multiple physical tables exist, the SQL Engine writes SQL against the tables required to retrieve the result set for a query.

MicroStrategy supports application-level partitioning for fact tables through one of two methods:

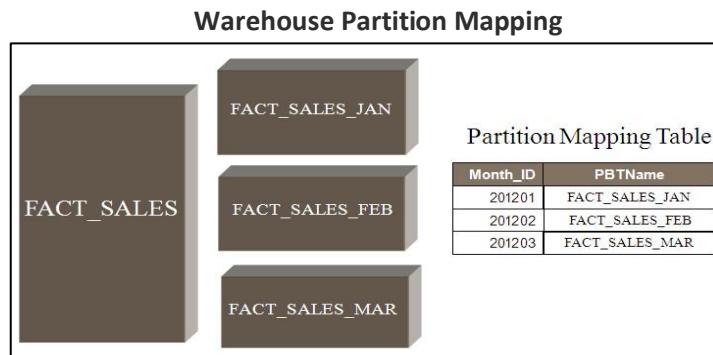
- **Warehouse partition mapping** is based on a physical warehouse partition mapping table (PMT) that resides in the data warehouse and describes the partitioned base tables that are part of a conceptual whole.
- **Metadata partition mapping** is based on rules logically defined in Architect and stored in the metadata.

Warehouse partition mapping

Warehouse partition mapping uses a special table that maps attribute IDs to corresponding partition tables in the data warehouse.

Instead of adding the original fact table or the partition base tables in the project, you create and maintain a partition mapping table, which Architect uses to identify the partitioned base tables as part of a logical whole. You only bring the partition mapping table into the project.

To understand how you can use warehouse partition mapping to manage partitioned base tables, consider the following example. To ease maintenance and improve query performance, you have divided a multibillion-row fact table by month into a few one-billion-row fact tables. The following illustration shows a partition mapping table that points to partitioned fact tables in the data warehouse:



To use a partition mapping table in Architect, implement the following conventions:

- You can use any name for the partition mapping table.
- The table must have a column for each attribute ID that the table is partitioned by. These attribute IDs represent the partitioning attributes.

You use ID columns to represent partitioning attributes.

- The table must include a PBTName column that contains the names of the partition base tables. Partition base tables are the actual physical partition tables that make up the complete, logical fact table.

The PBTName column indicates to Architect that the table is a partition mapping table.

- A row for each of the partition base tables must exist in the logical whole.

When you follow these conventions, Architect automatically recognizes the partition mapping table.

Implementing warehouse partition mapping

When you use warehouse partition mapping, you add the partition mapping table to the project in Architect. You do not add the partition base tables to the project. When you add the partition mapping table, it automatically associates this table with the underlying partition base tables. After adding the partition mapping table to the project, you must also create a partition map to identify the attribute used to partition the tables.

To use warehouse partition mapping in a project

1. In Developer, in the appropriate project, open Architect.
2. In the Warehouse Tables pane, in the list of tables in the database instance, select the partition mapping table, right-click, and select **Add Table to Project**.
3. Click **Save and Close**.

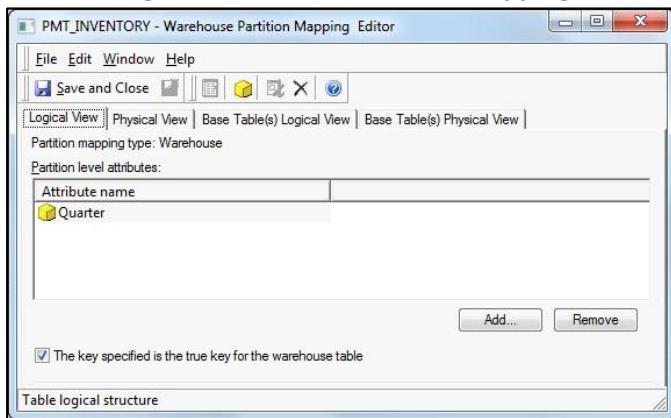
*When you add a partition mapping table to a project, you see a message window that reminds you to set the partitioning attribute level in the partition mapping. Click **OK**.*

4. In Developer, in the Schema Objects folder, in the Partition Mappings folder, double-click the partition mapping object with the same name as the partition mapping table you just added to the project.
5. In the Warehouse Partition Mapping Editor, on the Logical View tab, click **Add**.
6. In the Partition Level Attributes Selection window, from the **Available Attributes** list, select the attribute that the tables are partitioned by and click the > button to add it to the **Selected Attributes** list.
7. Click **OK**.
8. In the Warehouse Partition Mapping Editor, click **Save and Close**. Then update the project schema.

The Warehouse Partition Mapping Editor has four tabs, each displaying different information about the partition mapping:

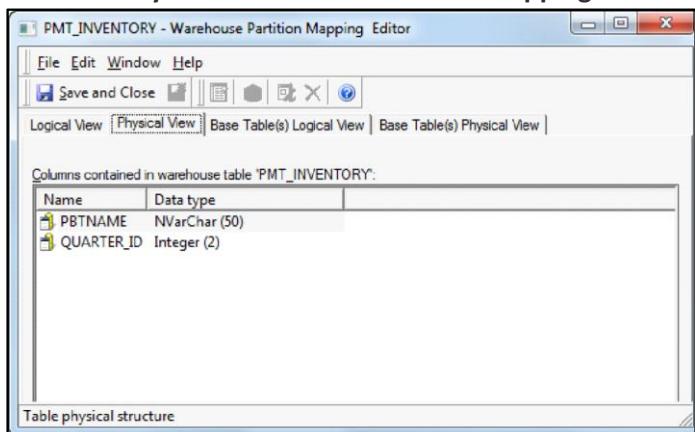
- **Logical View tab** shows the attribute by which the base tables are partitioned. In the image below, the base tables are partitioned by the Quarter attribute.

Logical View of the Partition Mapping



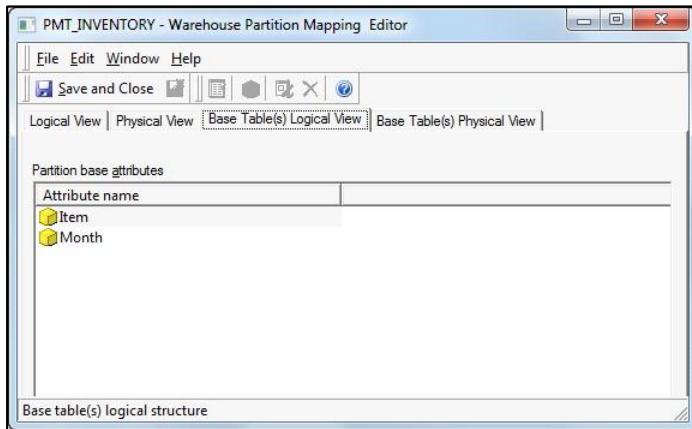
- **Physical View tab** shows the actual columns in the partition mapping table. In this example, the partition mapping table contains two columns, PBTNAME and QUARTER_ID.

Physical View of the Partition Mapping



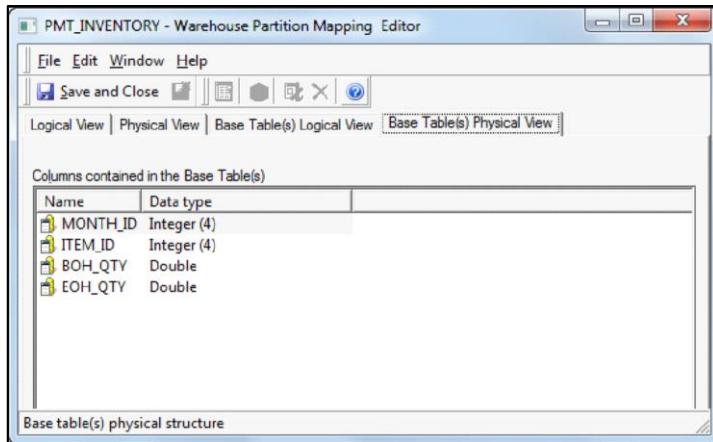
- **Base Tables Logical View tab** shows the attributes mapped to the base partition tables. The following image shows the logical view of the associated partition base tables. In this example, the Item and Month attributes are mapped to the base partition tables:

Logical View of the Partition Base Tables



- **Base Tables Physical View tab** shows the actual columns in the partition base tables. The following image shows the physical view of the base partition tables. In this example, the Month attribute is mapped to the MONTH_ID column, and the Item attribute is mapped to the ITEM_ID column in the base partition tables. In addition, you can map the Beginning on Hand and End on Hand facts to the BOH_QTY and EOH_QTY columns, respectively:

Physical View of the Partition Base Tables



When you run a report that requires information from one of the partition base tables, the Query Engine first runs a prequery to the partition mapping table to identify the partition that contains the requested data. The prequery requests the partition base table names associated with the attribute IDs from the filtering criteria. Next, the SQL Engine generates SQL against the appropriate partition base tables to retrieve the report results.

The partition base tables may contain either the same column as the partitioning attribute or a column corresponding to a child of the partitioning attribute.

Architect does not support application-level partitioning for lookup tables. The size of lookup tables usually makes this kind of partitioning unnecessary. You can use server-level partitioning on lookup tables.

Metadata partition mapping

Metadata partition mapping uses a partition mapping object stored in the metadata to specify the data contained in the partition base tables.

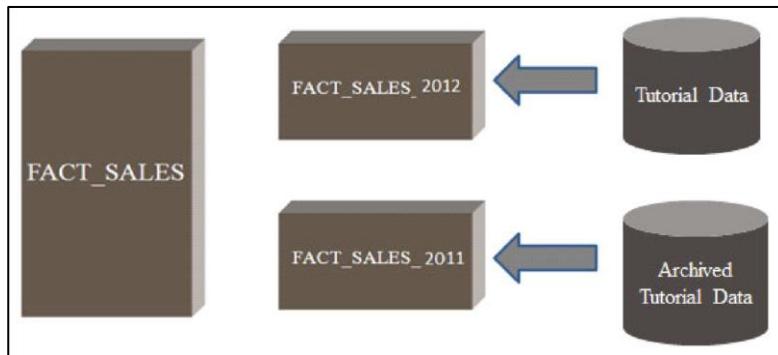
When you execute a report that runs against the partitioned tables, the Query Engine sends the necessary prequeries to the metadata to identify the partition base tables to include in the report SQL. The SQL Engine then generates SQL against the appropriate partition base tables to retrieve the report results.

Implementing metadata partition mapping

To implement metadata partition mapping, you add the partition base tables to the project in Architect. Next, you create the partition mapping and define the data slices for each partition base table.

With metadata partition mapping, you can also add partition base tables from multiple data sources to create a single partition definition. This feature can improve performance, especially in high data volume projects.

For example, all 2012 partition base tables are in Tutorial Data, but the 2011 partition base tables are in Archived Tutorial Data. You can add the 2011 and 2012 partition base tables from both data sources to a project and define the data slice for each partition base table. The following diagram shows a metadata partition mapping table from multiple data sources:



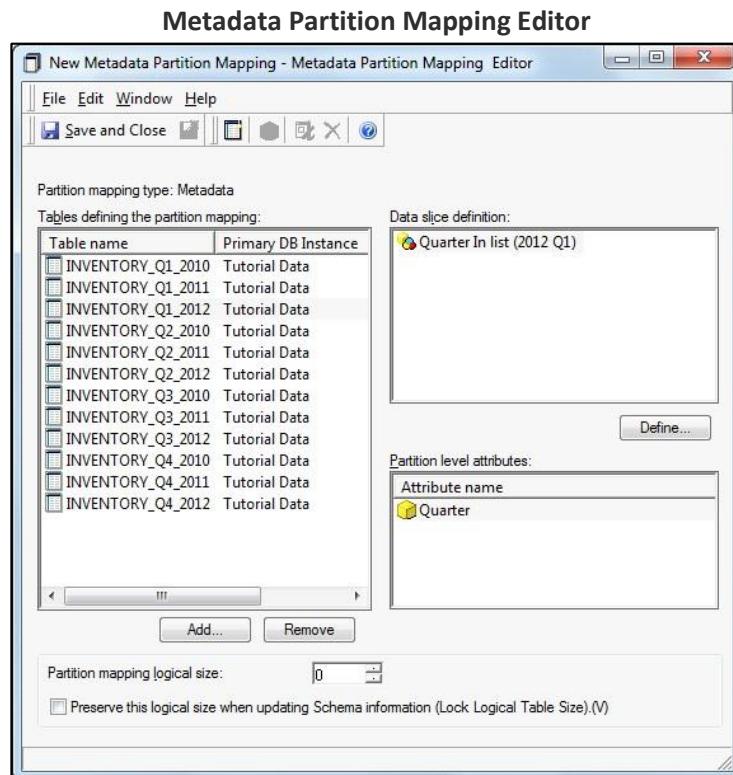
To use metadata partition mapping in a project

1. In Developer, in the appropriate project, open Architect.
2. In the Warehouse Tables pane, from the list of tables available in the database instance, select all the partition base tables. Right-click your selection and select **Add Table to Project**.
3. Click **Save and Close**.
4. In Developer, in the Schema Objects folder, select the **Partition Mappings** folder.
5. On the **File** menu, point to **New** and select **Partition**.
6. In the Partition Tables Selection window, from the **Available Tables** list, select the partition base tables and click the **>** button to add them to the **Selected Tables** list.

7. Click **OK**.

Define the partition base table

8. In the Metadata Partition Mapping Editor, from the Tables Defining the Partition Mapping list, select a partition base table.



9. Click **Define**.

10. In the Data Slice Editor, in the Data Slice Definition pane, define the data contained in the partition base table.

This step is similar to defining a filter condition.

11. In the Data Slice Editor, click **Save and Close**.

12. Repeat the steps for each partition base table in the partition mapping.

13. If you want to define a logical size for the partition mapping, in the Metadata Partition Mapping Editor, in the **Partition mapping logical size** box, type the desired value.

14. Click **Save and Close**. Name and save the partition mapping, then update the project schema.

Partitioning in In-Memory Analytics

MicroStrategy In-Memory Analytics combines a massive parallel in-memory data store with a state of the art visualization and interaction engine. You can process large volumes of information when the source data is divided into partitions. Each partition can have up to 2 billion rows.

In-Memory Analytics enables you to import multiple tables to store raw data in memory for quick access. You can interact with this imported data to create visualizations, reports, and dossiers.

In-Memory Analytics delivers high performance by partitioning and processing data in parallel. As with other distributed processing systems, you must identify the optimal partition attribute to develop a successful implementation.

Partitioning In-Memory Analytics cubes

Partitioning distributes data across multiple cores on a single box or across available CPU cores in the environment. Each data partition is in a “shared nothing” architecture and is assigned to a specific CPU core or group of cores.

Partitioned data can be aggregated using distributive functions such as SUM, MIN, MAX, COUNT, and PRODUCT. You can also rewrite semi-distributive functions such as STD DEV or VARIANCE as distributive functions. Scalar functions such as Add, Greatest, Date/Time Functions, String manipulation functions, and so on are also supported.

You can also create DISTINCT COUNT functions on the partition attribute, and create derived metrics using any of the functions supported by MicroStrategy.

Picking a partition attribute and the number of partitions

To partition the data that you import using In-Memory Analytics, select a single partition attribute for the entire dataset. All tables that include the partition attribute will have their data distributed along the elements of that attribute.

In-Memory Analytics supports all flavors of INT data types, STRING/TEXT, as well DATE data types for partitioning. INT data is distributed using either a MOD or HASH scheme, and TEXT/DATE data is distributed using a HASH scheme.

Select the partition attribute based on your specific application needs. To do this, use the following guidelines:

- Scan the largest fact tables you are importing to identify common attributes.
- Data should be partitioned to allow for the largest number of partitions to be involved in a given query.
- Do not select an attribute that is frequently used for filtering or selections, as they tend to push the analysis towards specific sets of partitions and minimize the benefits of parallel processing.
- A partition attribute should allow for near uniform distribution of data across partitions, so that the workload on each partition is evenly distributed.
- Columns that are used to join large tables make good partition attributes.

Data import

MicroStrategy enables analysts to import data from a variety of sources into projects. This feature provides a simple method for end users to access data that is not already included in your project.

The following examples are common connectors you can use to import data:

- File from Disk and File from URL: import delimited text files with extensions other than .csv. You can also import multiple spreadsheets, workbooks, or text files into a single Intelligent Cube.
- Clipboard: copy and paste data directly into the MicroStrategy interface.
- Database: import multiple relational tables into a single Intelligent Cube.
- Public Data: import data from Internet sources based on search terms.
- Dropbox (requires Dropbox account).
- Google Big Query (requires Google account).
- Google Drive (requires Google account).
- Google Analytics (requires Google Analytics account and configuration).
- Facebook (requires Facebook account).
- Twitter (requires Twitter account).
- Search Engine Indices (Apache Solr Search): import data based on search terms.
- Hadoop: import from Hadoop leveraging the MicroStrategy Big Query Engine; you can also import directly from Hadoop.
- BI Tools (SAP, SAP BO, Cognos): import data via web services URLs.
- OLAP Sources/MDX.
- MicroStrategy Project: import reports from other MicroStrategy projects.

Search Engine Indexes and BI Tools connectors load data directly from the source to the report, document, or dossier in MicroStrategy.

This lesson provides a brief overview of two common data import sources - databases and OLAP/MDX sources.

Picking database tables to build MTDI Cubes

You can simultaneously import two or more tables, Excel sheets, Excel workbooks, or CSV files from any supported data source. As part of the In-Memory Analytics Intelligent Cube design, you can also load tables into MicroStrategy without building a SQL query for the source database.

With the In-Memory Analytics cube architecture, you can import multiple tables from a database, and specify a partitioning attribute (distribution key) along with the number of distributions. Because MicroStrategy supports single node architecture, the number of distributions must match the number of CPU cores available on the MicroStrategy Intelligence Server machine.

OLAP/MDX sources

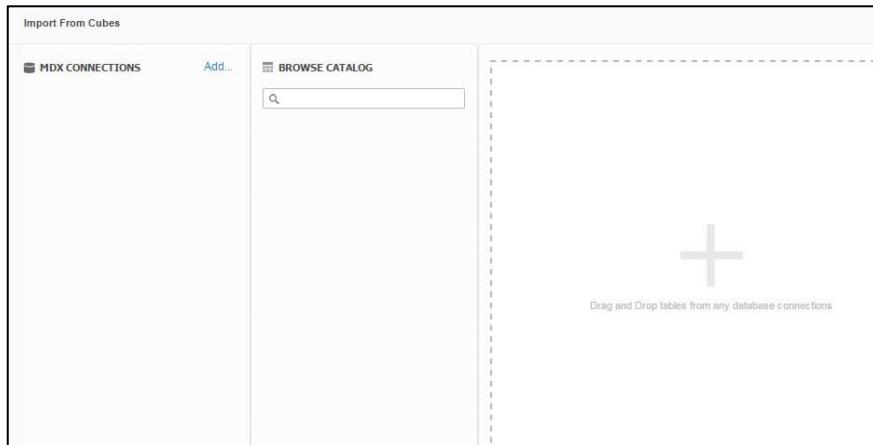
Data Import enables you to retrieve data from Online Analytical Processing (OLAP) and Multidimensional Expressions (MDX) sources. This type of data import is performed via Direct Data Access, so the data cannot be transferred into memory. The following OLAP/MDX sources are supported:

- IBM Cognos TM1
- Microsoft Analysis Services 2005/2012
- Oracle Essbase 9.3/11.x MDX Cube provider
- SAP BW & SAP BW 7.x

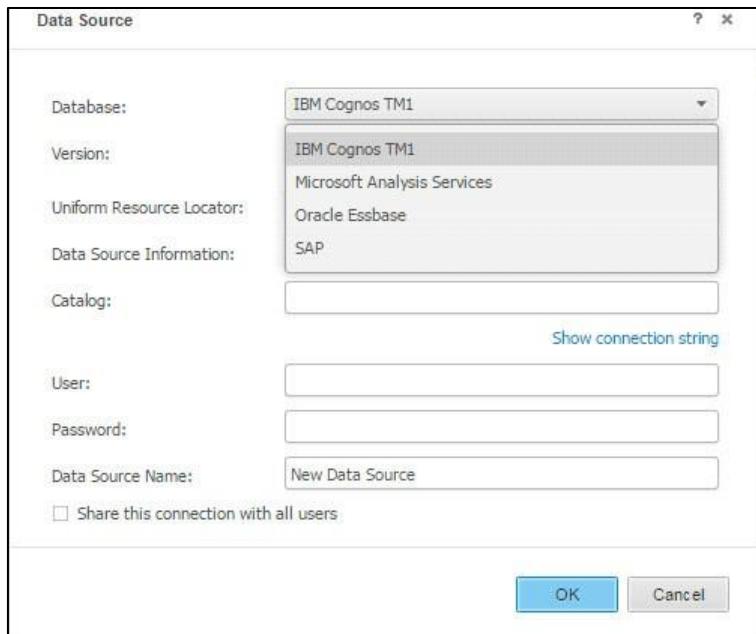
The steps below are not to be performed as an in-class exercise. They are for reference only.

To configure data import sources

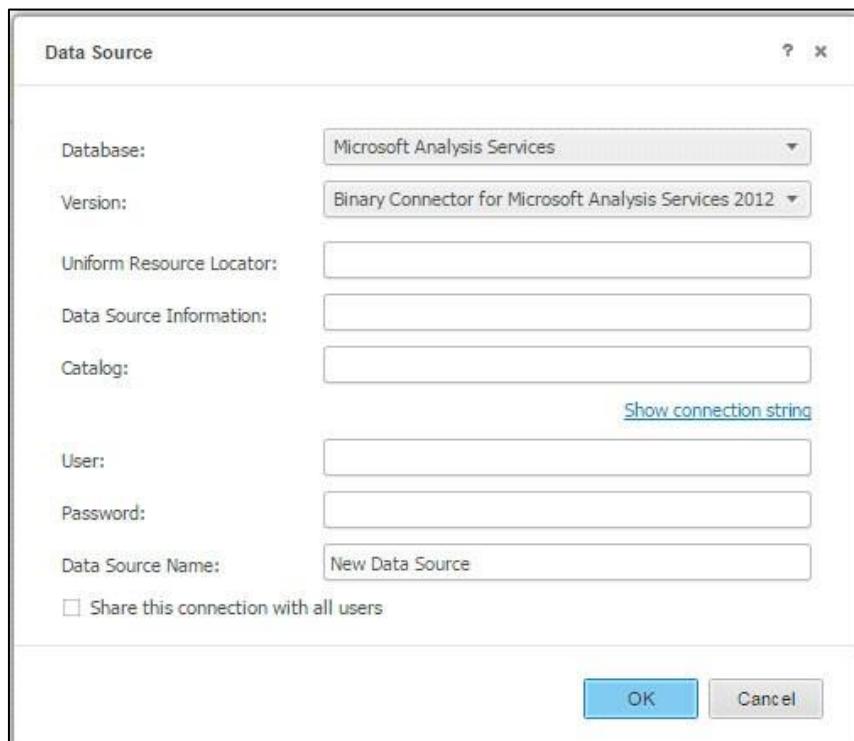
1. Click the **Add External Data** or **OLAP** icon in the Connect to Your Data window.

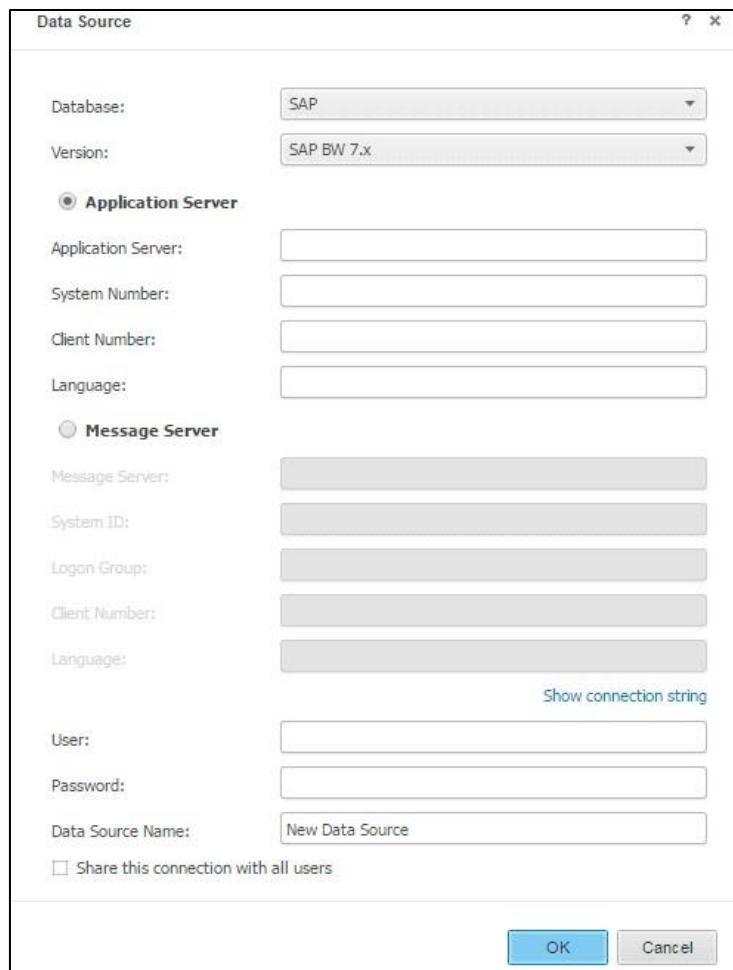


2. Click **Add** next to MDX Connections.



3. Choose one of the MDX sources.
4. Enter connection information for the selected MDX source. Below are some examples of MDX data source selections and the configuration parameter require



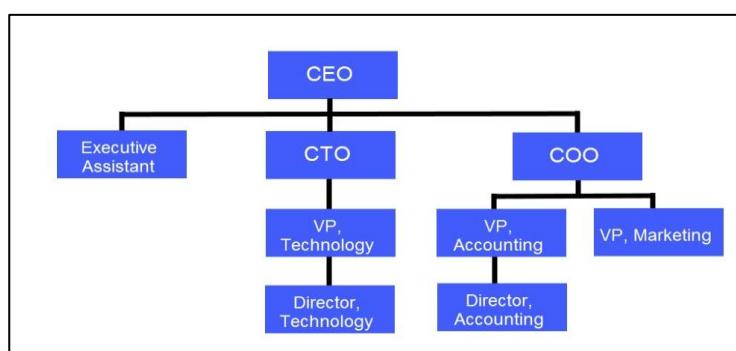


Once you are connected to the data source, you can select the desired objects to import into the project.

Importing hierarchical attributes for hierarchy reporting

Hierarchy reporting enable analysts to display data for all levels in a given hierarchy in dossiers. This is especially useful when displaying data for unbalanced hierarchies, which contain branches with inconsistent levels.

For example, in an Employee hierarchy, each division of the organization may contain a distinct number of managerial levels, as in the following example.



Notice that the Executive Assistant is only managed by the CEO, while the Director of Accounting is managed by The VP of Accounting, the COO, and the CEO. Hierarchy reporting enables analysts to create reports that clearly display the relationships between these managerial levels within the hierarchy.

Analysts can implement hierarchy reporting to view data on all levels of the hierarchy, and expand or collapse individual levels as desired. Hierarchical attributes can also be used to create visualizations.

To enable analysts to use hierarchy reporting in Dossiers, you import a hierarchical attribute that contains all levels of a hierarchy. For example, a hierarchical attribute for the Time hierarchy includes Year, Quarter, Month, and Day levels.

A hierarchical attribute is a new object that represents an entire hierarchy and its levels. Hierarchical attributes can be placed on a grid, filtered, and sorted, much like normal attributes. By treating an entire hierarchy as a single object, MicroStrategy makes it quick and easy to build hierarchy reports.

You can create hierarchical attributes in your projects when you import data from Microsoft Analysis Services or Oracle Essbase. Analysts can use your dataset to create a dossier that leverages the hierarchical attribute, as displayed in the following image.



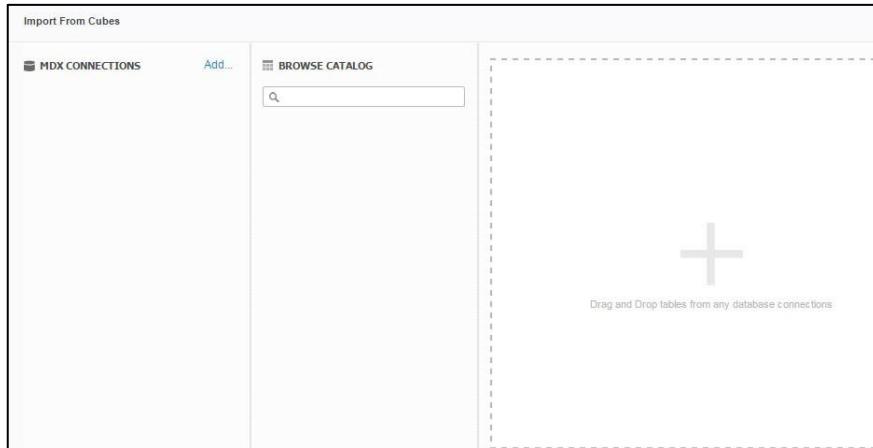
When you create a hierarchical report, you can customize it in the following ways:

- Create derived metrics with pass-through expressions performed directly in the data source.
- Create a custom sort pattern if traditional sort patterns do not fit your needs.
- Add security filters to ensure users can only view information that pertains to them.

The steps below are not to be performed as an in-class exercise. They are for reference only.

To create a hierarchical attribute

1. In MicroStrategy Web, click **Create** and select **Add External Data**.
2. In the Connect to Your Data window, click **OLAP**. The Import from Cubes window is displayed.



3. Under **MDX Connections**, select the Microsoft Analysis Services or Oracle Essbase data source connection that contains the desired data.
4. Click and drag the desired cube from the **Browse Catalog** pane to the right pane.
5. In the right pane, select the **Import hierarchy as hierarchical attribute** check box.
6. Click **Finish**.

Exercises

Exercise 11.1: Pick tables to build MTDI cubes

Multi-Table Data Import Cubes enable you to create in-memory cubes by directly importing data from a variety of sources.

In this exercise, you will:

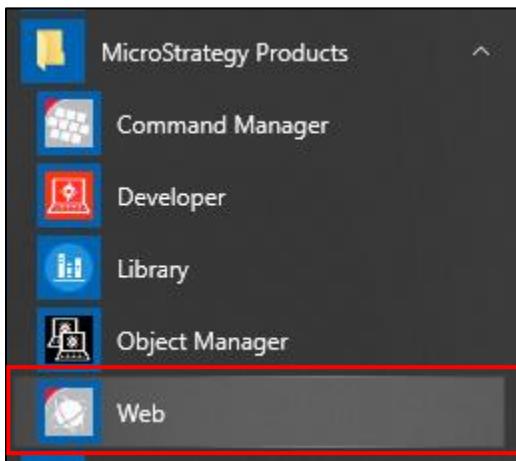
1. In MicroStrategy Web, access the MicroStrategy Tutorial project.
2. Import multiple relational tables.
3. Partition a fact table using a partitioning attribute (distribution key).

4. Create an MTDI cube with the In-Memory Analytics Intelligent Cube structure.
-

Import the data

Access MicroStrategy Web

1. On the Windows machine, click the Windows menu on the task bar.
2. Click **MicroStrategy Products**, and then click **MicroStrategy Web**.



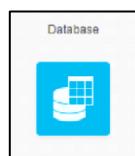
3. Select MicroStrategy Tutorial project

If prompted on the MicroStrategy Web Login page, use the following credentials:

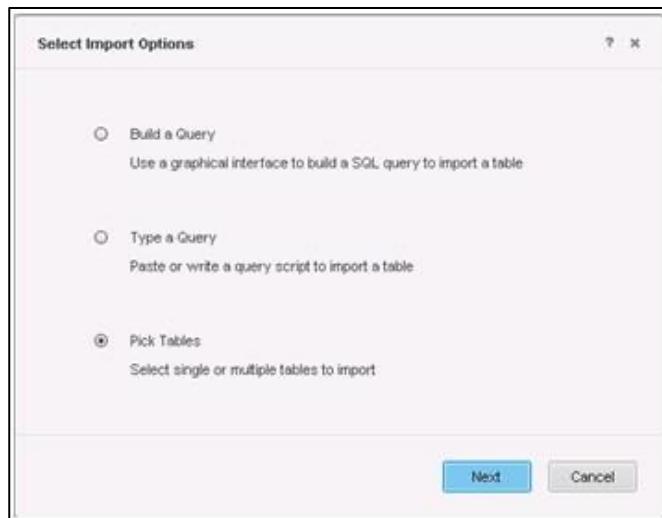
- *login id: administrator*
- *password: (none)*

Pick tables to build MTDI cubes

4. In MicroStrategy Web, click Create, and then click **Add External Data**. The Connect to Your Data window opens.
5. Click the **Databases** icon.

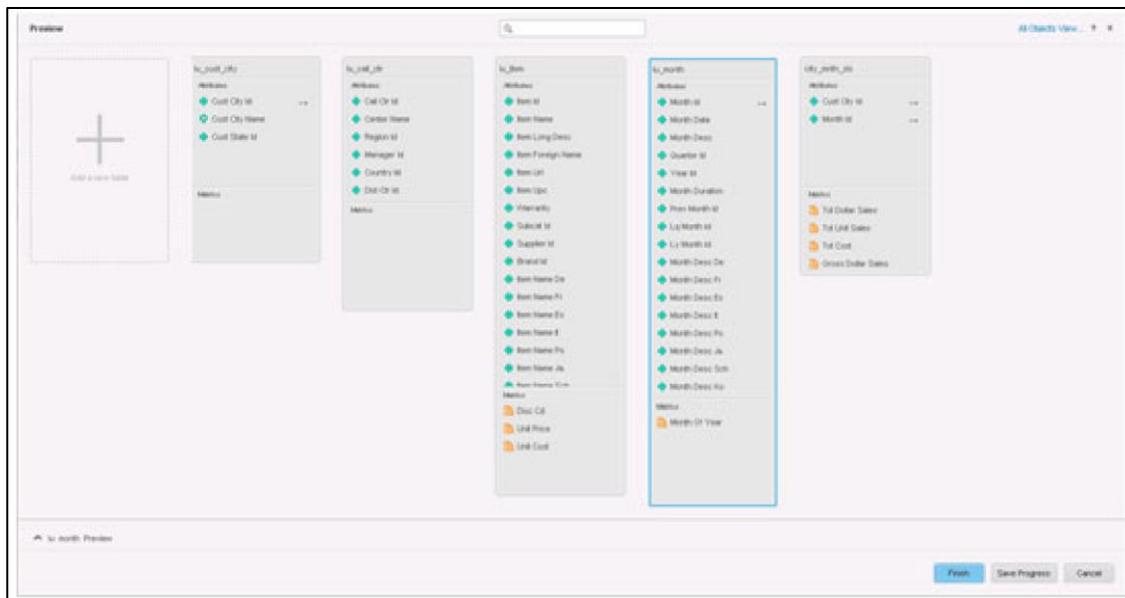


6. Select the **Select Tables** option and click **Next**.



7. Under Data Sources, select the **Tutorial Data** database instance.
8. In the Namespace drop-down list, select **tutorial_wh** and then select and drag the following tables to the right pane:
 - city_mnth_sls
 - lu_call_ctr
 - lu_cust_city
 - lu_item
 - lu_month
9. Click **Prepare Data**.

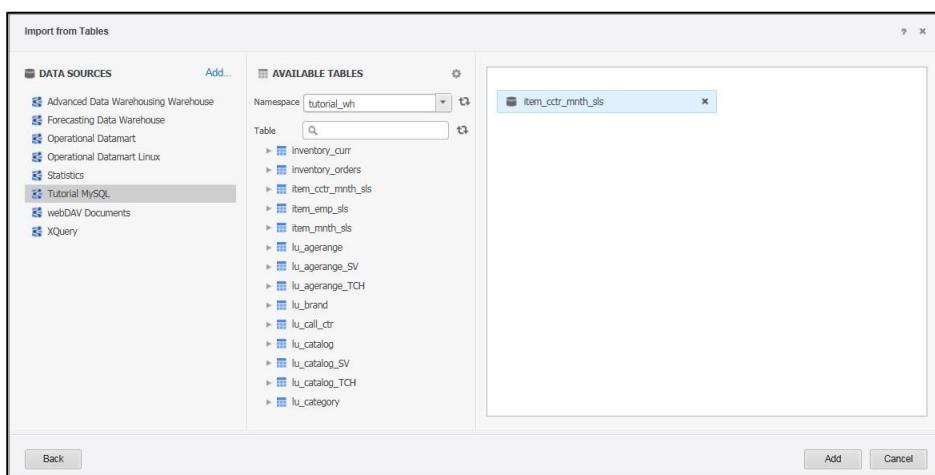
The Preview window shows all tables selected in the previous screen, and the respective columns mapped as attributes and metrics. Data Import automatically maps the table columns as attributes and metrics. Any errors in the attribute or the metric mappings will be corrected later in this section of the workshop.



Add an additional table

You can add additional tables after bringing in the initial set of tables using Data Import. You will now add the **item_cctr_mnth_sls** fact table. This table contains sales data at the item, call center, and month levels.

1. To add the table, click **Add a new Table** in the Preview window.
2. In the Connect to Your Data window, click the **Databases** icon.
3. Select the **Select Tables** option and click **Next**.
4. Under Data Sources, select the **Tutorial Data** database instance.
5. Drag the **item_cctr_mnth_sls** table to the right pane.



6. Click **Add**. The new table appears in the Preview window (you may need to scroll to the right to view the table).

The screenshot shows the Data Import Preview window with two tables side-by-side:

- city_mnth_sls**:
 - Attributes**: Cust City Id, Month Id
 - Metrics**: Tot Dollar Sales, Tot Unit Sales, Tot Cost, Gross Dollar Sales
- item_ccctr_mnth_sls**:
 - Attributes**: Call Ctr Id, Month Id, Item Id
 - Metrics**: Tot Dollar Sales, Tot Unit Sales, Tot Cost, Gross Dollar Sales

Create a multi-form attribute

7. In the Preview window, notice that each column in a given table will be imported as a separate attribute. For example, Call Ctr Id and Center Name are each listed as distinct attributes in the lu_call_center table.

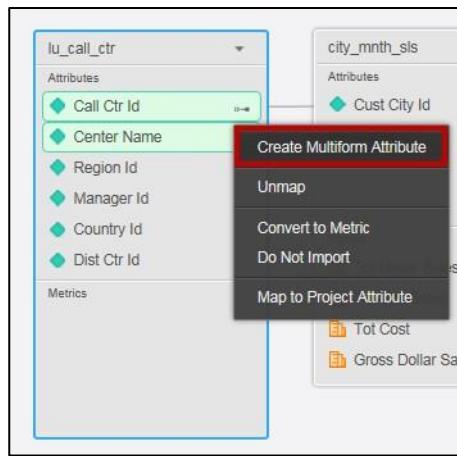
The screenshot shows the Data Import Preview window for the lu_call_ctr table. The Attributes section lists several columns as distinct attributes:

- Call Ctr Id
- Center Name
- Region Id
- Manager Id
- Country Id
- Dist Ctr Id

The "Call Ctr Id" and "Center Name" items are highlighted with a red box.

Data Import offers the opportunity to designate two or more columns as attribute forms within a single attribute.

8. To create a unified CALL CENTER attribute:
- Hold the **Ctrl** key and select both **Call Center Id** and **Center Name**.
 - Right-click the selection, then select **Create Multiform Attribute**.



- c. Change the attribute name to **CALL CENTER**.
- d. Ensure that the form (such as **Call Center Id**) and form categories (such as **ID**) are correctly matched.
- e. Clear the **Display Form** check box for Call Center ID to ensure that only the Call Center DESC appears by default on reports and documents.
- f. Click **Submit** to create the new multiform attribute.

The dialog box is titled 'Create Multi-forms Attribute'. It has a 'New Attribute Name' field containing 'CALL CENTER'. Below it is a dropdown menu set to 'lu_call_ctr'. The main area shows two rows for mapping attributes:

	Form Category	Display Form
Call Ctr Id	ID	<input type="checkbox"/>
Center Name	DESC	<input checked="" type="checkbox"/> <input type="button" value="X"/>

At the bottom are 'Submit' and 'Cancel' buttons.

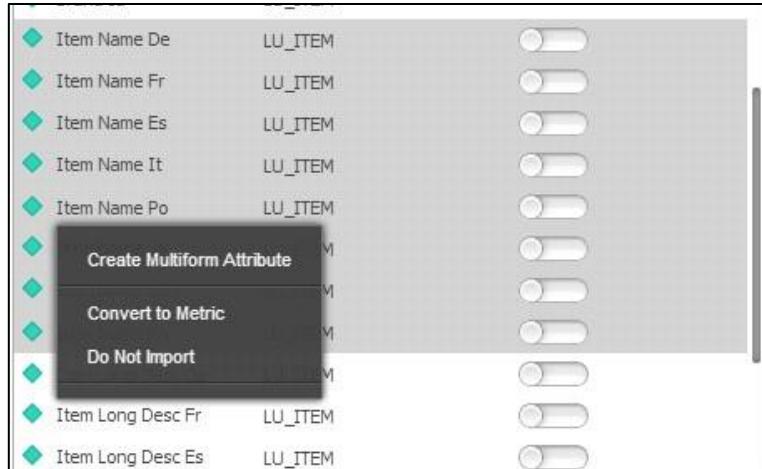
The lu_call_ctr table should now resemble the following in the Preview window:



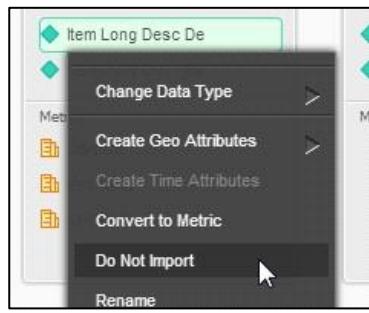
The name of the attribute will change in all other tables containing at least one of the attribute forms. For example, the Call Ctr Id column in all other tables will also change to CALL CENTER.

9. Repeat the steps for the following columns in the respective lookup tables:
 - a. CUSTOMER CITY (Cust City Id, Cust City Name)
 - b. ITEM (Item Id, Item Name)
 - c. MONTH (Month Id, Month Desc)
10. Click **All Objects View** to see all attributes and metrics in a list.
11. Right-click the following metrics and select **Do Not Import**:
 - a. Disc Cd
 - b. Gross Dollar Sales
12. Do not import any extraneous attribute columns that were not used in the multiform attributes created above.

*To select multiple columns at the same time, hold the **Shift** key, and then click the first and last column in the range of desired columns.*



You can also right-click a given column and select **Do Not Import** in the Preview window.



13. In the All Objects View window, convert Month of Year from a metric to an attribute. To do this, right-click the **Month of Year** metric and select **Convert to Attribute**.

You can also convert a metric to an attribute (and vice versa) in the Preview window.

14. The metric and attribute names are automatically populated according to the respective database column names. In some cases, these names may not accurately reflect the metric's data to the end user. To rename the Tot Unit Sales > Unit Sales metric, in the All Objects View window, right-click the metric and select **Rename**.

Keep the All Objects window open.

Establish the Partition Attribute in the All Objects View window.

Ideally, you should partition against the largest fact tables available. Because an in-memory cube can only be partitioned on a single attribute, you should choose an attribute that exists in both fact tables. In this case, the Item attribute meets this criteria.

You can validate the selection of the Item attribute by examining the distribution of Item elements across the row count of each fact table. The item_cctr_month_sls has the following distribution:

Item	Metrics	Rowcount
1		448
2		474
3		520
4		440
5		434
6		441
7		450
8		481
9		493
10		480
11		445
12		483

In the image above, you can see that the elements of Item are fairly evenly distributed throughout the fact table (examining the rest of the returned data confirms this).

Item also has a fairly even distribution in the city_mnth_sls table.

Choose 2 for the number of partitions because the training machines typically come with 2 CPU cores. This number should be adjusted based on the number of cores in the Intelligence Server machine. Match the number of cores to the number of partitions to fully leverage the in-memory analytics parallel processing functionality.

Establish the partition attribute and finish the import

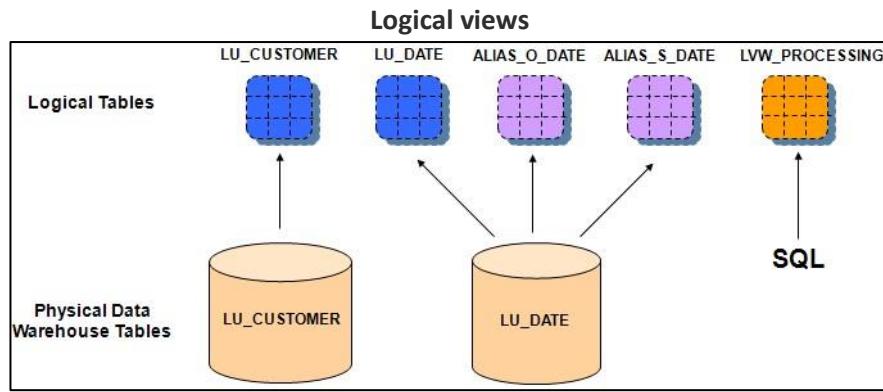
1. To partition the ITEM attribute, do the following:
 - a. In the All Objects View window, from the **Partition Attribute** drop-down list, select **ITEM**.
 - b. In the **Number of Partition** box, enter **2**.
2. Click **Close** to return to the Preview window.
3. Click **Finish**.
4. Click **Import as an In-memory Dataset**.
5. Save the cube in your My Reports folder as **MTDI Workshop Cube**.

You can now use this MTDI cube as a data source for a dossier or document.

CHAPTER 12: LOGICAL VIEWS

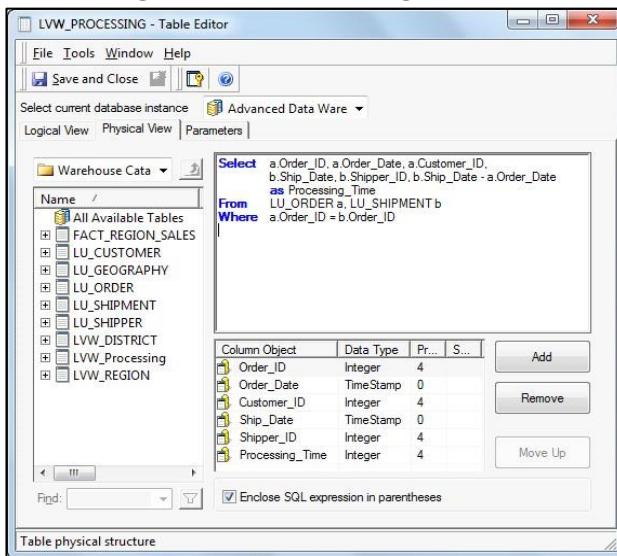
A logical view is a logical table that you create in your project by using a SQL query that is executed against your data warehouse. Logical views are an alternative to database views that you create and maintain at the application level. For example, you can create a logical view to develop an attribute form expression that includes a calculation based on multiple tables in the data warehouse.

For example, order and shipping information are stored in the LU_ORDER and LU_SHIPMENT tables in the data warehouse. Analysts want to calculate the processing time between the order dates in the LU_ORDER table and the ship dates in the LU_SHIPMENT table to analyze the processing time for each order. To satisfy this requirement, you can create a logical view that performs the desired calculation:



In this example, the LVW_PROCESSING logical table is a logical view that is created by running a SQL query against the LU_ORDER and LU_SHIPMENT tables in the data warehouse. It does not map directly to any physical table. When you define the logical view, you write a SQL statement that selects the pertinent information from the LU_ORDER and LU_SHIPMENT tables and then calculates the difference between the order and ship dates to determine the processing time. The following example shows this logical view in the Logical Table Editor:

Logical Table Editor—Logical View



The logical view contains a SQL statement and definitions for each column that is created as part of the logical view. Map the attributes and facts to these columns.

Other methods for creating schema objects

This appendix provides information on alternative methods for creating schema objects using the Project Creation Assistant and editors. It also defines the sort order for user hierarchies using the Hierarchy Editor.

Project Creation Assistant and editors

You first create a project object in the Project Creation Assistant. Then you can use a combination of wizards and editors to create schema objects, as an alternative to creating the project and schema objects in Architect.

The following Project Creation Assistant wizards and editors are available:

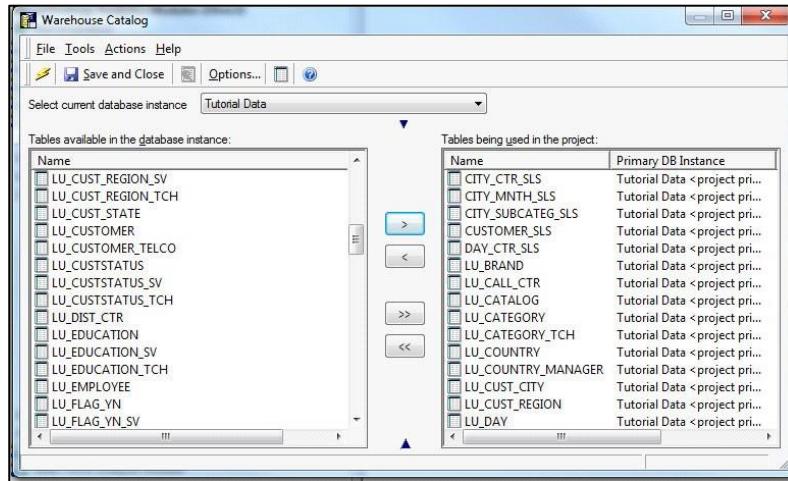
- Warehouse Catalog
- Fact Creation Wizard
- Fact Editor
- Attribute Creation Wizard
- Attribute Editor
- Hierarchy Editor

Creating schema objects using the Project Creation Assistant involves the following high-level steps:

- Select the project tables using the Warehouse Catalog:** In this interface, you first select the database instance associated with the project, and then select the tables.

The following example shows the Warehouse Catalog with lookup and fact tables added to a project:

Warehouse Catalog: Lookup and Fact Tables Added to Project



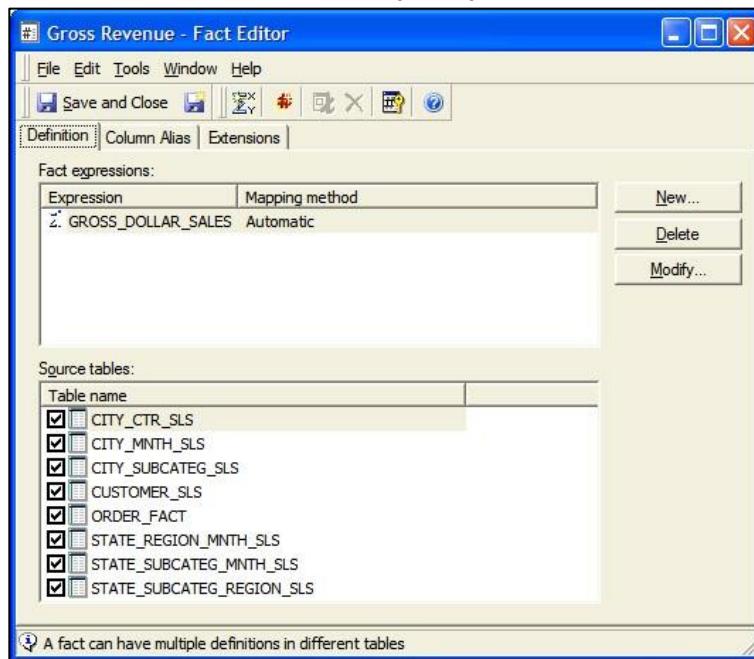
- Create facts:** The Fact Creation Wizard helps you create facts. You define the fact creation rules and then create facts using this wizard. The following example shows the Fact Creation Wizard with facts created in a project:

Fact Creation Wizard: Facts Created in Project



The following example shows the Fact Editor with a simple expression for the Gross Revenue fact:

Fact Editor: Simple Expression



3. **Create attributes:** The Attribute Creation Wizard enables you to create multiple, basic attributes quickly. In this wizard, you define the creation rules, create ID and description forms, select lookup tables, create attribute relationships, and create compound attributes. You can use Attribute Editor to create more complex attributes. The following example shows the Attribute Creation Wizard with selected attributes created in a project:

Attribute Creation Wizard: Attributes Created in Project



The following example shows the Attribute Creation Wizard with the correct description forms selected for each attribute:

Attribute Creation Wizard: Description Forms Selected

Attribute Creation Wizard - Description Column Selection

Select the column that will represent the description for each attribute below

Attributes:	
Attribute name	Description column name
Customer City	CUST_CITY_NAME
Customer Region	CUST_REGION_NAME
Customer State	CUST_STATE_NAME
Customer	CUST_LAST_NAME
Education	EDUCATION_DESC
Gender	GENDER_DESC
Income	BRACKET_DESC
Month	MONTH_DESC
Order	Use ID as description
Payment Type	PYMT_DESC
Quarter	QUARTER_DESC
Ship Date	Use ID as description
Year	Use ID as description

Buttons: Help | Cancel | < Back | Next > | Finish

The following example shows the Attribute Creation Wizard with the correct lookup tables selected for each attribute:

Attribute Creation Wizard: Lookup Tables Selected

Attribute Creation Wizard - Lookup Table Selection

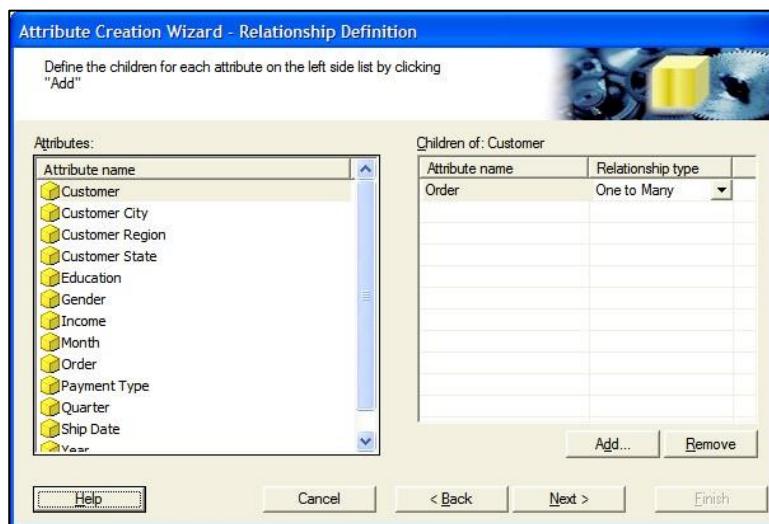
Select the (lookup) table that will contain the description for each attribute below

Attributes:	
Attribute name	Lookup Table Name
Customer City	LU_CUST_CITY
Customer Region	LU_CUST_REGION
Customer State	LU_CUST_STATE
Customer	LU_CUSTOMER
Education	LU_EDUCATION
Gender	LU_GENDER
Income	LU_INCOME
Month	LU_MONTH
Order	ORDER_FACT
Payment Type	LU_PYMT_TYPE
Quarter	LU_QUARTER
Ship Date	ORDER_FACT
Year	LU_YEAR

Buttons: Help | Cancel | < Back | Next > | Finish

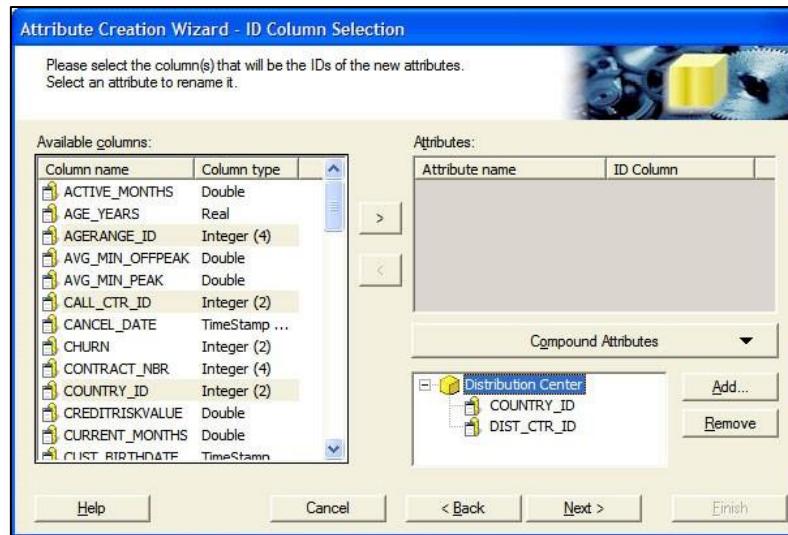
The following example shows the Attribute Creation Wizard with the relationships created for an attribute:

Attribute Creation Wizard: Relationships Created

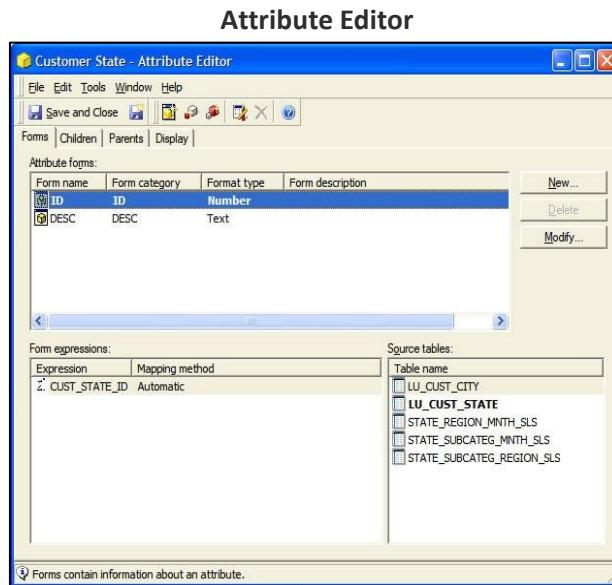


The following example shows the Attribute Creation Wizard with a compound attribute created:

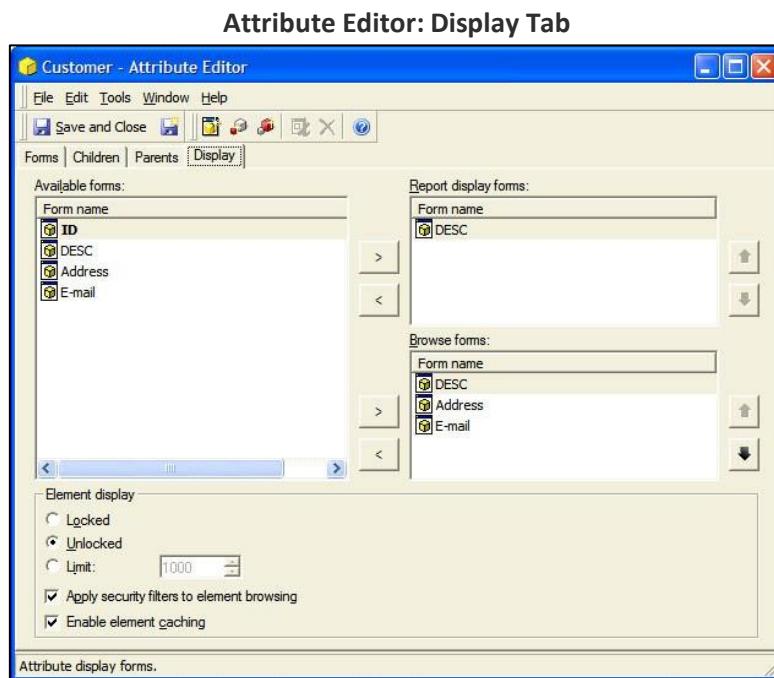
Attribute Creation Wizard: Compound Attribute Created



The following example shows the Attribute Editor:

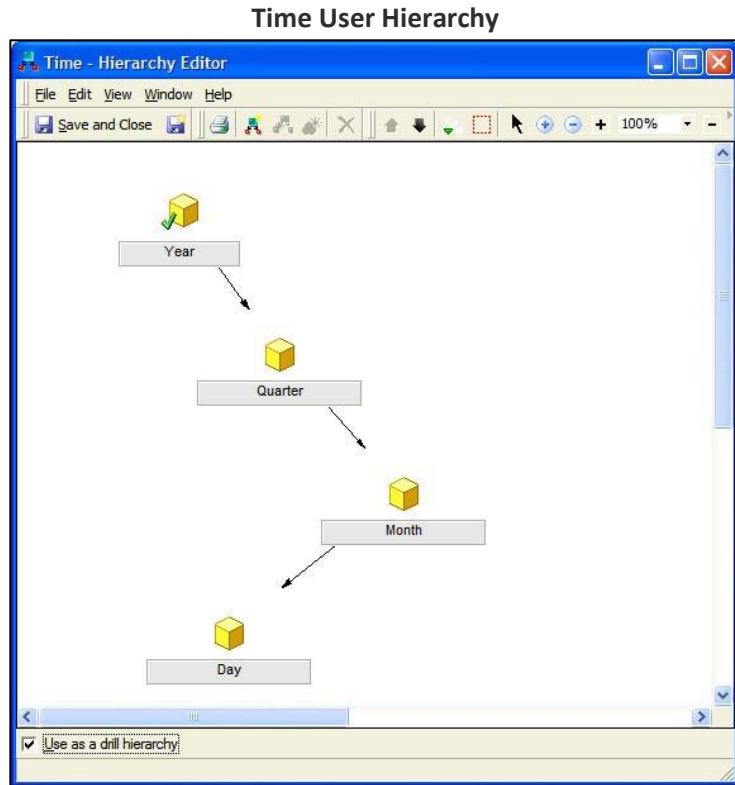


The following example shows the Display tab on the Attribute Editor. It is useful to define the Report and Browse elements of attributes:



4. **Create user hierarchies:** You can create and modify user hierarchies using the Hierarchy Editor. It enables you to configure a variety of hierarchy-related settings.

The following example shows the Hierarchy Editor with the Time user hierarchy, which includes the Year, Quarter, Month, and Day attributes:



Defining the sort order for user hierarchies

When you create user hierarchies, you can define the sort order used to display their attributes. You use the Project Configuration Editor and the Hierarchy Editor to define the displayed sort orders for browsing attributes and drilling on attributes.

Defining the sort order for browsing

By default, when you create a user hierarchy for browsing, its attributes display in alphabetical order. However, you can customize the order in which the attributes display. For example, it may make more sense to users for the attributes to display in a logical order based on their relationships.

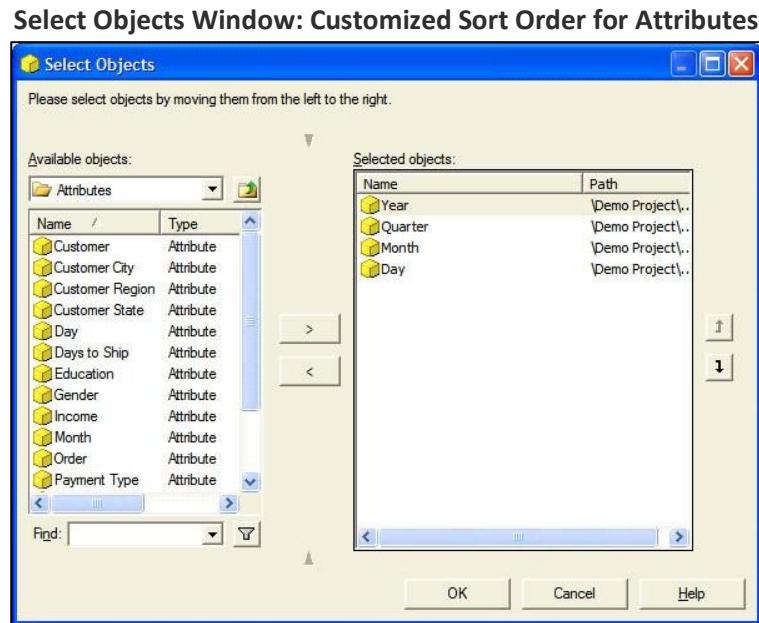
The sort order that you define for browsing also applies to hierarchy prompts, since these objects are based on browsing user hierarchies.

To define the sort order for a browsing user hierarchy

1. In Developer, right-click the project that contains the user hierarchy to define the sort order for, and select **Project Configuration**.

2. In the Project Configuration Editor, under the Project definition category, select **Advanced**.
3. Under Advanced Prompt Properties, clear the **Display Attribute alphabetically in hierarchy prompt** check box.
4. Click **OK**.
5. In the Schema Objects folder, in the Hierarchies folder, in the Data Explorer folder, right-click the user hierarchy to define the sort order for, and select **Edit**.
6. In the Hierarchy Editor, from the **Edit** menu, select **Add Attributes**.
7. In the Select Objects window, use the arrow buttons to the right of the Selected Objects list to change the order of the attributes in the user hierarchy to the desired display order.

The following image shows the Select Objects window with a customized sort order for the attributes in the Time user hierarchy:



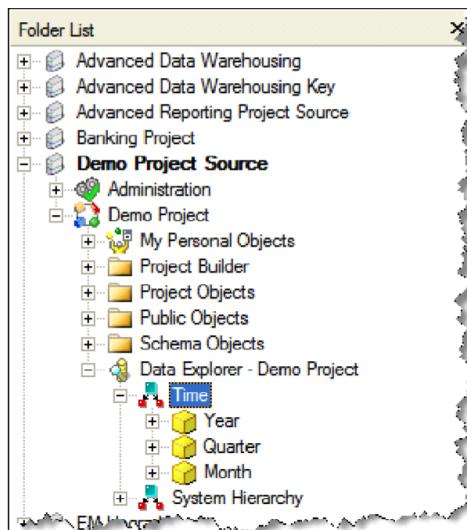
The attributes in the Time user hierarchy are ordered from highest to lowest rather than alphabetically.

If you clear the **Display Attribute alphabetically in hierarchy prompt** project-level setting, the order of the attributes in the Selected Objects list determines the order in which they display in the user hierarchy. Click **OK** to exit the Select Objects window.

8. In the Hierarchy Editor, click **Save and Close**.
9. After closing the Hierarchy Editor, update the project schema.

When browsing, users now see the attributes displayed as shown in the following image:

Data Explorer: Customized Sort Order for Attributes



- In the image above, the Day attribute does not display because it is not an entry point for the user hierarchy.
- You may need to refresh the user hierarchy in the Data Explorer browser to see the customized sort order.

Defining the sort order for drilling

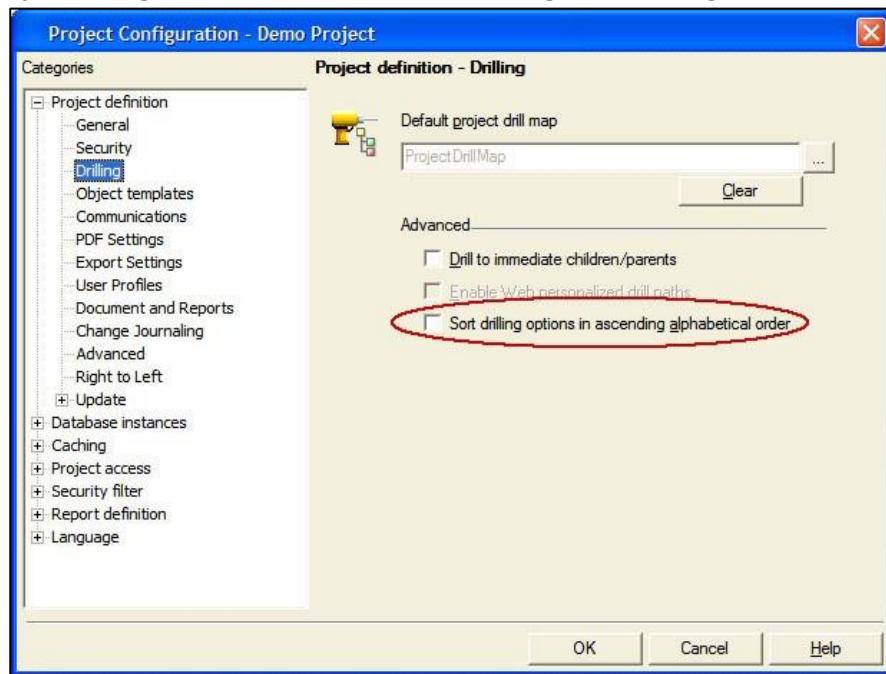
By default, when you create a user hierarchy for drilling, its attributes display according to their order in the Hierarchy Editor. If you do not customize the sort order of the attributes in the Hierarchy Editor, they display in alphabetical order. However, you can change the sort order to arrange the attributes in whatever order is most convenient for users when drilling on reports.

If you configure a user hierarchy for both browsing and drilling, you can only define one sort order. If you want the user hierarchy to display attributes using different sort orders depending on whether you use it for browsing or drilling, you must create separate user hierarchies for browsing and drilling.

To define the sort order for a drilling user hierarchy

1. In Developer, right-click the project that contains the user hierarchy to define the sort order for, and select **Project Configuration**.
2. In the Project Configuration Editor, under the Project definition category, select **Drilling**.

Project Configuration Editor: Sort Order Setting for Browsing User Hierarchies

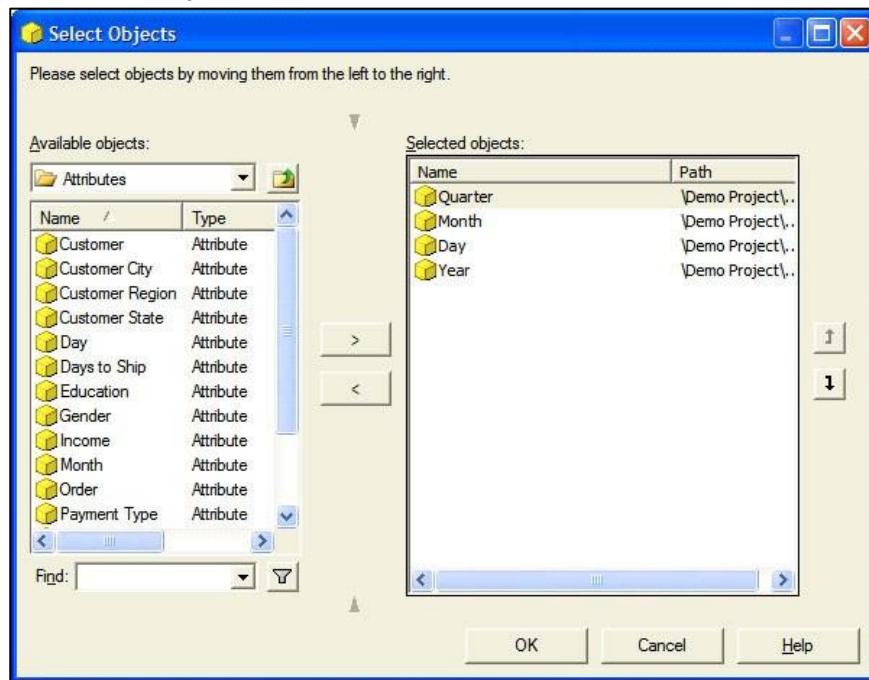


3. Under Advanced, clear the **Sort drilling options in ascending alphabetical order** check box if it is not already cleared.
4. Click **OK**.
5. In the Schema Objects\Hierarchies folder, right-click the hierarchy to define the sort order for, and select **Edit**.
6. In the Hierarchy Editor, from the **Edit** menu, select **Add Attributes**.
7. In the Select Objects window, use the arrow buttons to the right of the Selected objects list to change the order of the attributes in the user hierarchy to the desired display order.

After you clear the project-level setting, the order of the attributes in the Selected objects list determines the order in which they display in the user hierarchy.

The following image shows the Select Objects window with a customized sort order for the attributes in the Time user hierarchy:

Select Objects Window: Customized Sort Order for Attributes



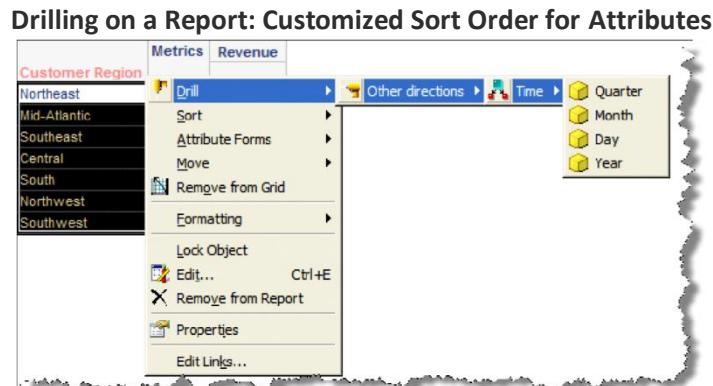
The attributes in the Time user hierarchy are ordered from the most to least frequently used attribute for drilling.

8. Click **OK**.

9. In the Hierarchy Editor, click **Save and Close**.

10. After closing the Hierarchy Editor, update the project schema.

When you view the user hierarchy while drilling on a report, the attributes display using this same sort order, as shown in the following image:

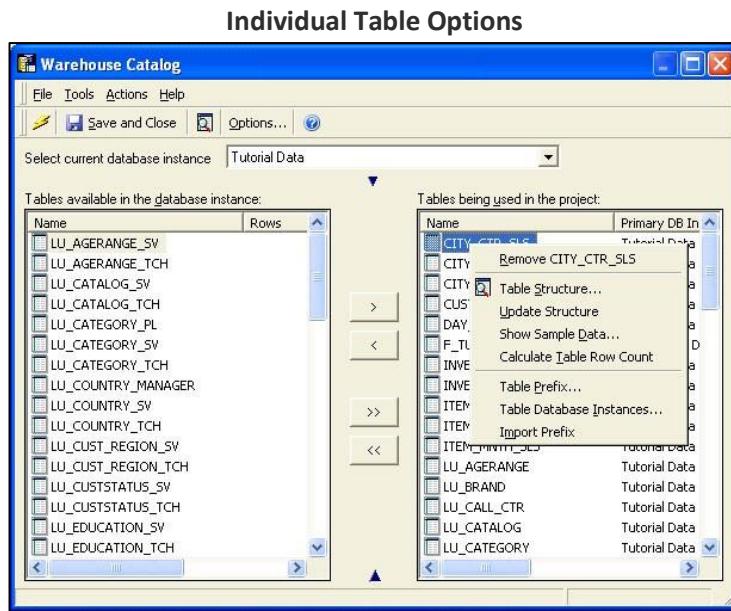


Warehouse Catalog

The Warehouse Catalog enables you to maintain the integrity of the logical tables with the data warehouse structure by providing a variety of options that apply to the project tables on an individual basis.

Maintaining individual tables

You access Warehouse Catalog options by right-clicking any table you have added to a project. The following image displays these individual table options:



The Warehouse Catalog provides the following options for individual tables:

- **Remove:** Remove a table from the project.
- **Table Structure:** View the columns and data types stored in a logical table in the Table Structure window. If the table structure has changed since you added the table to the project, you can click **Update Structure** to have Architect recognize the changes.
- **Update Structure:** A shortcut to the Update Structure option available in the Table Structure window.
- **Show Sample Data:** View the first 100 rows of data in a table.
- **Calculate Table Row Count:** Calculate and display the row count for a table.
- **Table Prefix:** Add and remove prefixes and assign a prefix to a table, in the Table Prefix window. Assigning a prefix to a table means that any time the SQL Engine uses that table, it includes the prefix when referencing the table.
- **Table Database Instances:** Select the primary database instance for a table and assign secondary database instances for a table, in the Available Database Instances window. Choose additional database instances in which you want to search for the table if you cannot obtain it from the primary database instance.

The primary database instance is the database instance that you should use for element browsing against the selected table and queries that do not require joins to other tables. The secondary database instance is any other database where the table also exists. You use it to support database gateways and when you create duplicate tables with MultiSource Option.

- **Import Prefix:** Retrieve a prefix for a table.

The MicroStrategy Tutorial data warehouse does not contain any table prefixes.

You can also automatically display the prefixes for all project tables in the warehouse catalog using a project-wide option in the Warehouse Catalog.

Project-wide Warehouse Catalog options

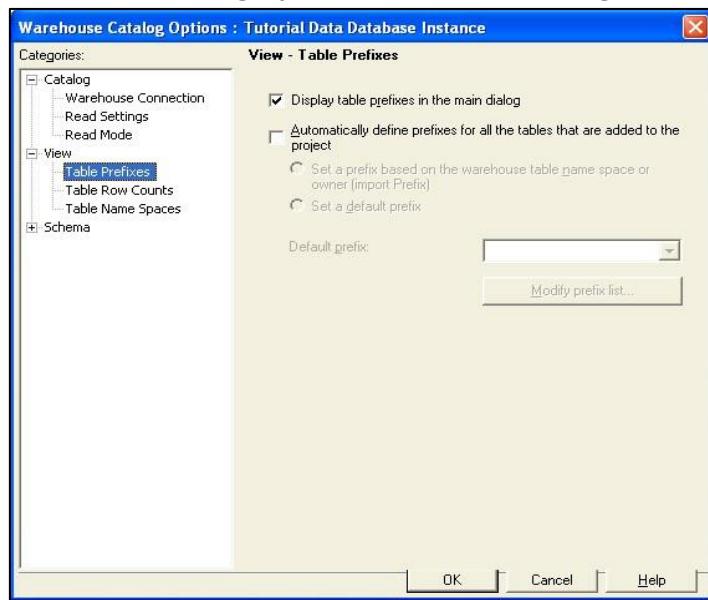
In addition to the table-level options, the Warehouse Catalog contains options that apply across a project. You access these options in the Warehouse Catalog by clicking **Options** on the toolbar. The following image displays project-wide options in the Warehouse Catalog Options window:



These project-wide options are grouped as follows:

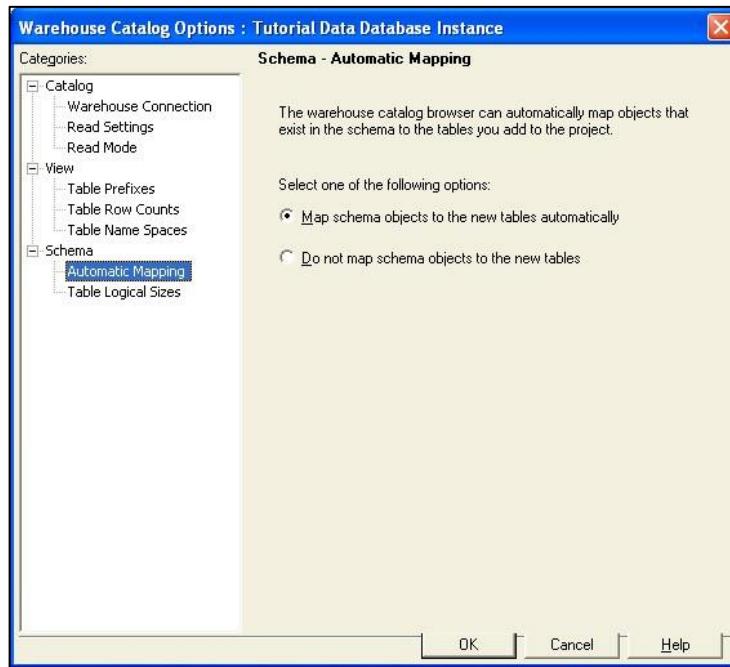
- **Catalog:** Quickly edit the primary project database instance, specify custom login, customize the SQL statement to query the database catalog tables, define different column reading settings, and specify the default read mode.
- **View:** Specify table viewing options, such as displaying table prefixes, row counts, and name spaces by default.

View Category in the Warehouse Catalog



- **Schema:** Configure automatic mapping of schema objects to the tables brought into the Warehouse Catalog and the calculation of logical table sizes.

Schema Category in the Warehouse Catalog



The Warehouse Catalog options provide flexibility in maintaining your project over time.

Adding tables with the MultiSource Option

You can add tables associated with secondary database instances using the Warehouse Catalog. The following steps show this process.

Add tables from a secondary database instance to a project in the Warehouse Catalog

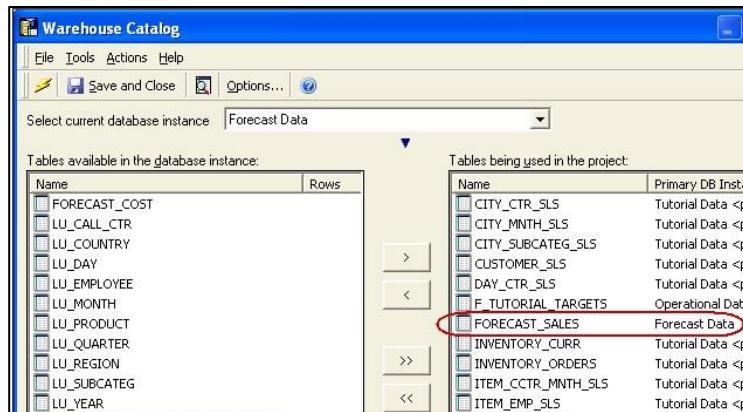
1. In Developer, open the project to add the tables to.
2. Open the Warehouse Catalog. In the **Select current database instance** drop-down list, select the database instance that contains the tables to add.
This drop-down list defaults to the primary database instance for the project.
3. In the **Tables available in the database instance** list, select the tables to add.
4. Click the **>** button to add the tables to the project.

The tables display in the **Tables Being Used in the Project** list, along with the primary database instance for each table.

5. Click **Save and Close**, then update the project schema.

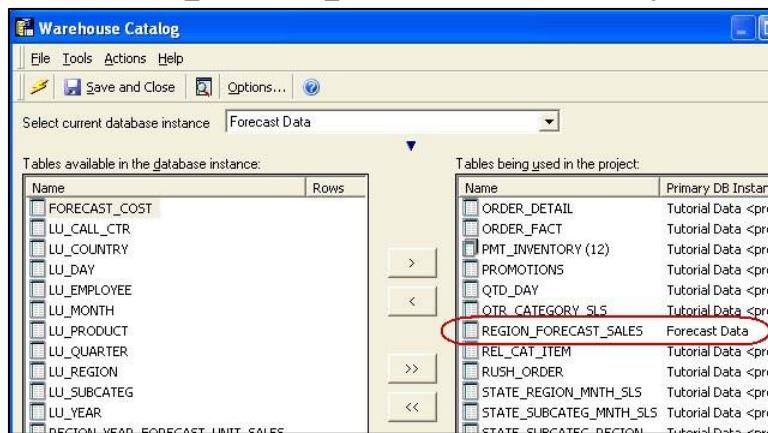
The following image shows the Warehouse Catalog with the FORECAST_SALES table added to the project:

FORECAST_SALES Table Added to Project



The following image shows the Warehouse Catalog with the REGION_FORECAST_SALES table added to the project:

REGION_FORECAST_SALES Table Added to Project



As you can see, the primary database instance for both of these tables is Forecast Data, not Tutorial Data.

To add duplicate tables to a project in the Warehouse Catalog

1. In Developer, open the project to add the tables to.
2. Open the Warehouse Catalog. In the **Select current database instance** drop-down list, select the database instance that contains the tables to add.
3. In the **Tables available in the database instance** list, select the tables to add.
4. Click the **>** button to add the tables to the project.

When you add a table that is already mapped to another database instance, a warning displays asking if you want to create a duplicate table and associate it with the selected database instance.

5. In the warning window, under Available Options, keep the **Indicate that <Table Name> is also available from the current DB Instance** option selected.

If you click the **Make no changes to <Table Name>** option, the table is not mapped to the selected database instance, and no duplicate table is created.



6. Do one of the following:

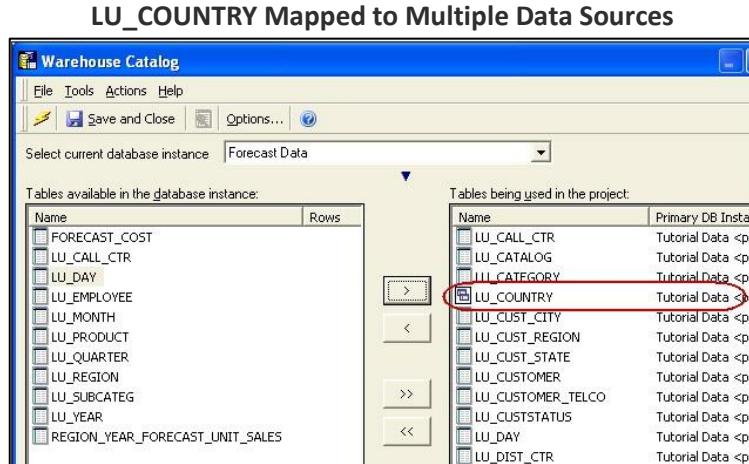
- To view and respond to the warnings for each duplicate table individually, click **OK**. In each successive warning window, click **OK** until no more warnings display.
- To respond to the warnings for all duplicate tables at the same time, click **OK for All**.

The tables display in the Tables Being Used in the Project list along with the primary database instance for each table. The icons beside the tables indicate that they are mapped to multiple data sources.

You can verify the database instances that a table is mapped to. In the Tables Being Used in the Project list, right-click the table to check and select **Table Database Instances**. The Available Database Instances window displays the primary and secondary database instances for the table.

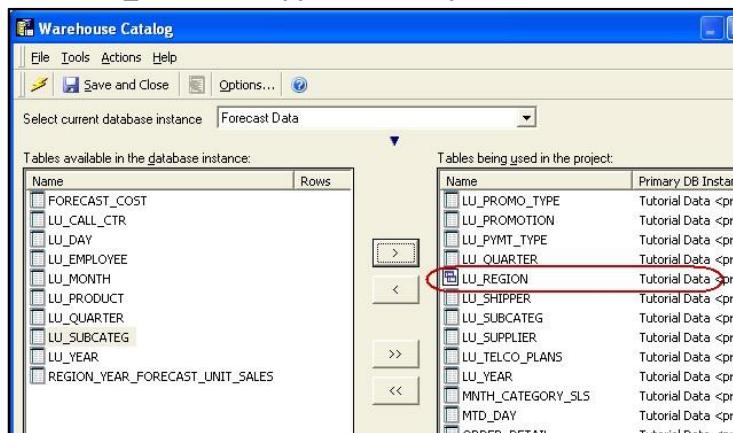
7. Click **Save and Close**, then update the project schema.

The following image shows the Warehouse Catalog with the LU_COUNTRY table mapped to multiple data sources:



The following image shows the Warehouse Catalog with the LU_REGION table mapped to multiple data sources:

LU_REGION Mapped to Multiple Data Sources

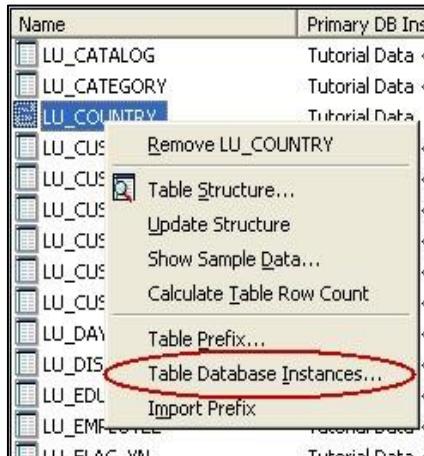


The primary database instance for both of these tables is Tutorial Data. However, the icons beside the tables indicate that they are also mapped to another database instance.

To change the primary database instance for a table in the Warehouse Catalog

1. In Developer, open the appropriate project.
2. Open the Warehouse Catalog. In the Tables Being Used in the Project list, right-click the table to change the primary database instance for, and select **Table Database Instances**.

Option for Accessing the Database Instances for a Table



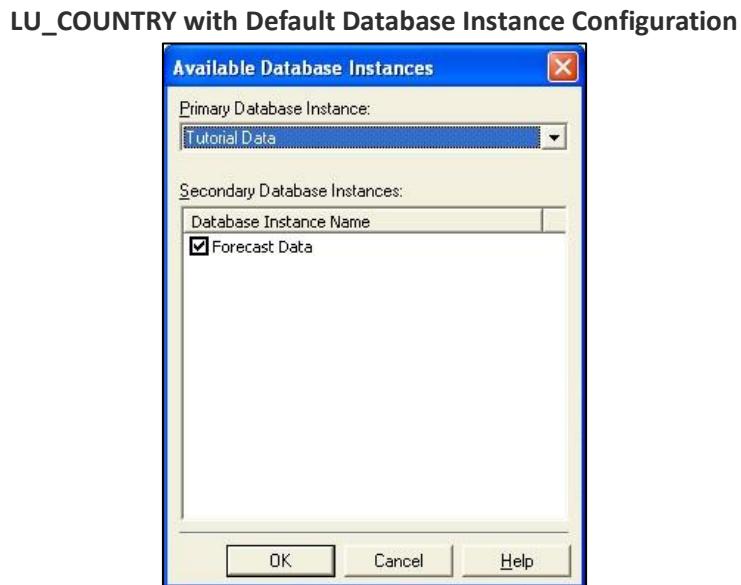
3. In the Available Database Instances window, in the **Primary Database Instance** drop-down list, select the database instance to use as the primary database instance.

The current primary database instance is moved to the Secondary Database Instances list.

4. To keep the current primary database instance as a secondary database instance:
 - a. In the **Primary Database Instance** drop-down list, select a new database instance.

- b. In the **Secondary Database instance** list, ensure the former primary database instance is selected.
5. Click **OK**.
6. In the Warehouse Catalog, click **Save and Close**, then update the project schema.

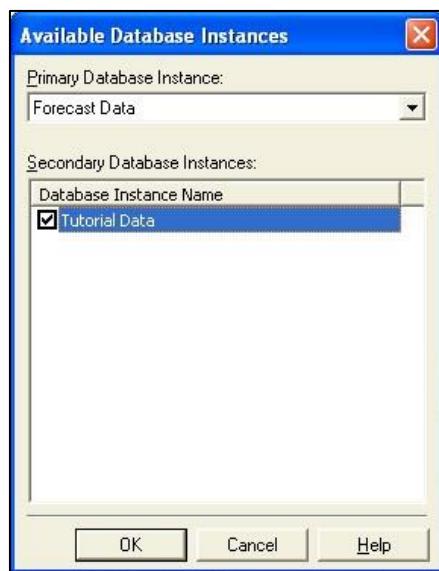
The following image shows the Available Database Instances window with the default database instance configuration for the LU_COUNTRY table:



Tutorial Data is the primary database instance for the LU_COUNTRY table, while Forecast Data is a secondary database instance for the table.

The following image shows the Available Database Instances window with the modified database instance configuration for the LU_COUNTRY table. Forecast Data is now the primary database instance for the LU_COUNTRY table, and Tutorial Data is a secondary database instance for the table.

LU_COUNTRY with Modified Database Instance Configuration



CHAPTER 13: ADVANCED ARCHITECTING

Many to Many Relationships

Review of Attribute Relationships

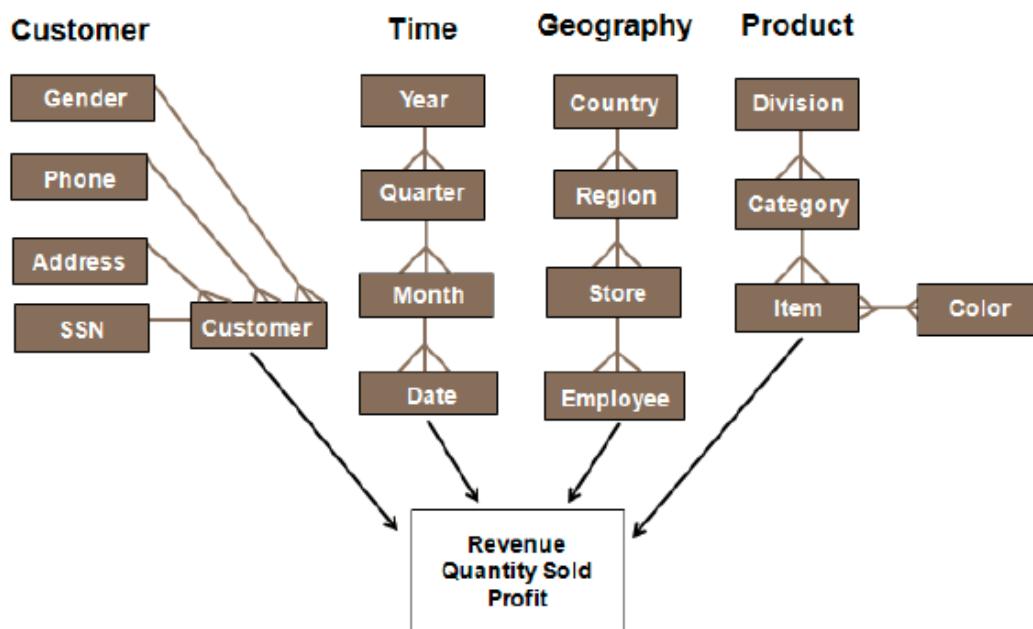
Attribute relationships are essential to the structure of a logical data model as they are the key to joining data from different attributes and aggregating fact data to different levels. Attribute relationships describe logical associations between attributes that exist based on business rules.

Attributes are related to each other in one of the following ways:

- Direct — A parent-child relationship exists between two or more attributes. In the data warehouse, direct relationships are explicitly defined using lookup tables or distinct relationship tables.
- Indirect — Two or more attributes are related only through a fact or set of facts. In the data warehouse, indirect relationships are stored only in fact tables.

The following logical data model shows a variety of direct and indirect attribute relationships:

Direct and Indirect Attribute Relationships

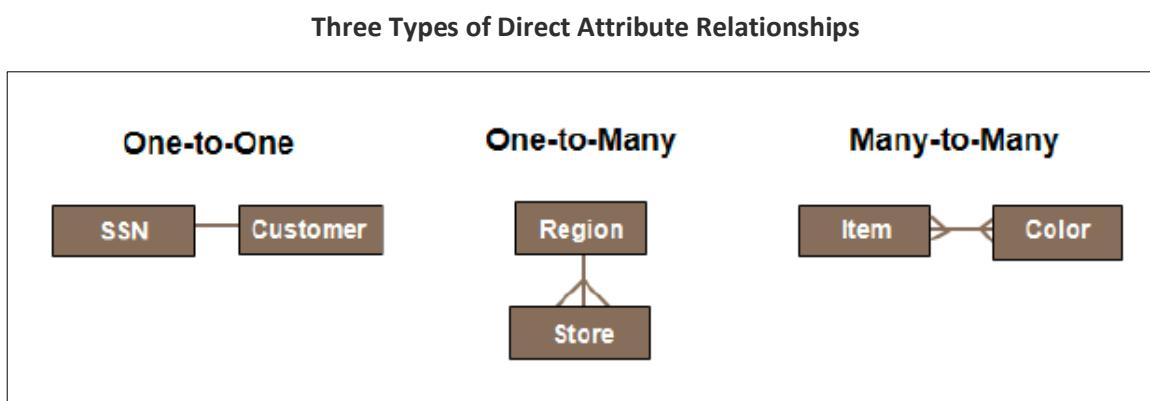


For example, Month and Date are attributes that have a direct relationship. Each month is directly related to specific dates. However, Customer and Date are attributes that have an indirect relationship. Without facts, these attributes are not inherently related, but if customers make purchases on specific dates, you can relate these two attributes through the facts that capture those sales. In this logical data model, you can relate Customer and Date using the Revenue, Quantity Sold, or Profit facts.

Directly related attributes have one of the following types of relationships:

- One-to-one
- One-to-many
- Many-to-many

The following illustration shows examples of these three relationship types:



The ways in which you model direct attribute relationships affect both the design of hierarchies in the logical data model and the structure of the data warehouse schema.

Attributes that have one-to-one or one-to-many relationships are very straightforward. You can easily relate these types of attributes using only lookup tables. However, attributes with many-to-many relationships are more complex. They require a separate relationship table to map the parent-child relationship. This table relates the lookup tables of the respective attributes.

Examples of Many-to-Many Relationships

- Student and Teacher—a student will have one or more teachers. A specific teacher will have many students.
- Doctor and Patient—in a hospital a doctor will be assigned to any number of patients. A specific patient may be assigned to one or more doctors.
- Course and Student—a college offers many courses and a specific course can be taken by one or more students.
- Product and Invoice—a product can appear on many invoices and an invoice can have many products.

Challenges of Many-to-Many Relationships

Because many-to-many relationships require distinct relationship tables, you have to design the logical data model and data warehouse schema in such a way that you can accurately analyze the relationship in regard to any relevant fact data.

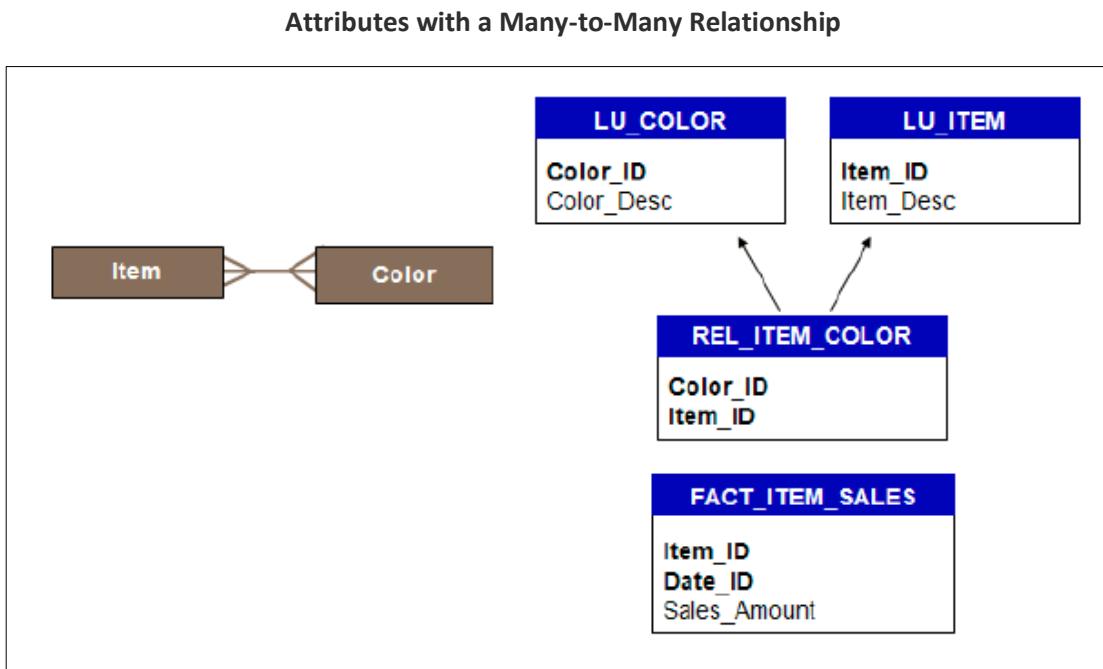
If the structure of your logical data model and data warehouse schema does not adequately address the complexities of querying attribute data that contains many-to-many relationships, you can have the following problems:

- Lost analytical capability
- Multiple counting

Lost Analytical Capability

When you have attributes with many-to-many relationships, you need to design the logical data model and data warehouse schema to ensure that users can answer any relevant questions about the data. Otherwise, users may lose the ability to analyze certain business scenarios.

In the following example, the Color and Item attributes have a many-to-many relationship:



An item can come in multiple colors, and the same color can apply to multiple items. The **LU_COLOR** and **LU_ITEM** tables store a distinct list of all colors and items respectively. The **REL_ITEM_COLOR** table enables you to join data in the two lookup tables, mapping the relationships between items and colors. The **FACT_ITEM_SALES** table stores sales by item and date.

In analyzing this relationship, users may want to know answers to the following questions:

1. In what colors are the various items available?
2. How much of a particular item and color combination was sold?

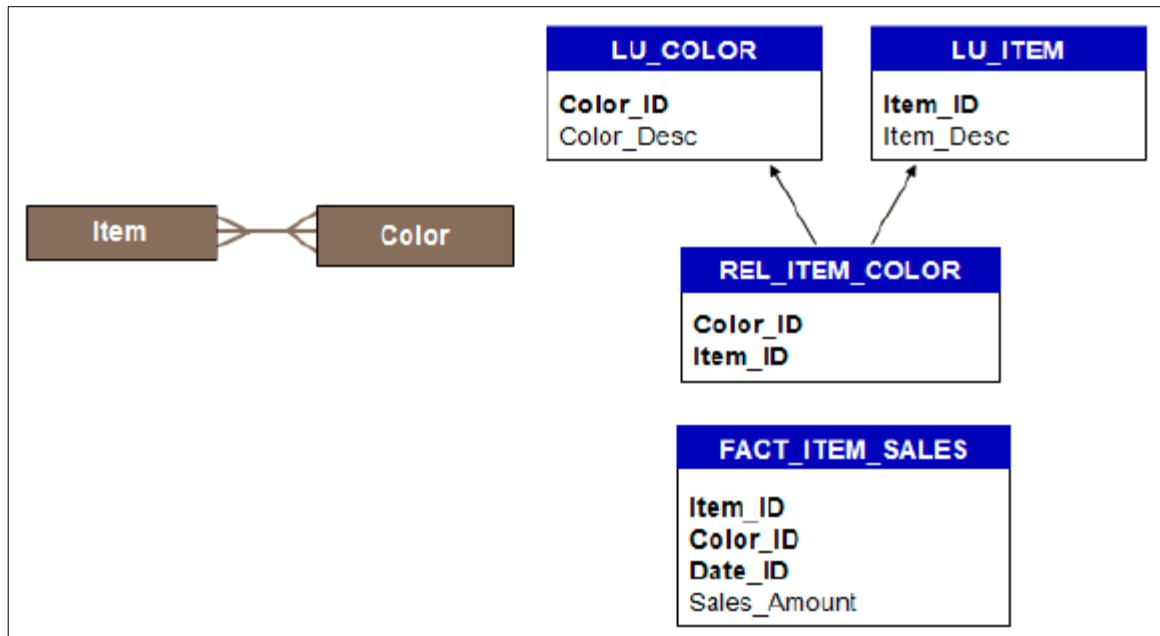
Answering the first question requires a table that contains a list of all possible item and color combinations. With many-to-many relationships, the distinct relationship table is what you use to map the relationship between two attributes. Therefore, the **REL_ITEM_COLOR** table contains a row for every possible item and color combination. The information in this table is sufficient to answer this first question. You can determine the colors in which an item is available.

The presence of the **REL_ITEM_COLOR** table is not sufficient to answer the second question. This relationship table only lets you analyze in which colors items are available, not which item and color combinations are

actually sold. Answering the second question requires a fact table that has sales information along with both item and color information. The FACT_ITEM_SALES table only contains the Item_ID column, not Color_ID. Therefore, it only stores which items sell, not the color of the items that sell. Because the color information is not part of the fact table, you cannot answer this second business question.

If you want to be able to analyze the sales of item and color combinations, you have to capture both the item and color data for sales in your source system. Then you have to modify the FACT_ITEM_SALES table to include both item and color information. The following illustration shows the same scenario, but the FACT_ITEM_SALES table now contains both the Item_ID and Color_ID columns:

Many-to-Many Relationship—Both Attributes in Fact Table



To ensure that you do not lose analytical flexibility when dealing with many-to-many attribute relationships, you need the following tables in your data warehouse schema:

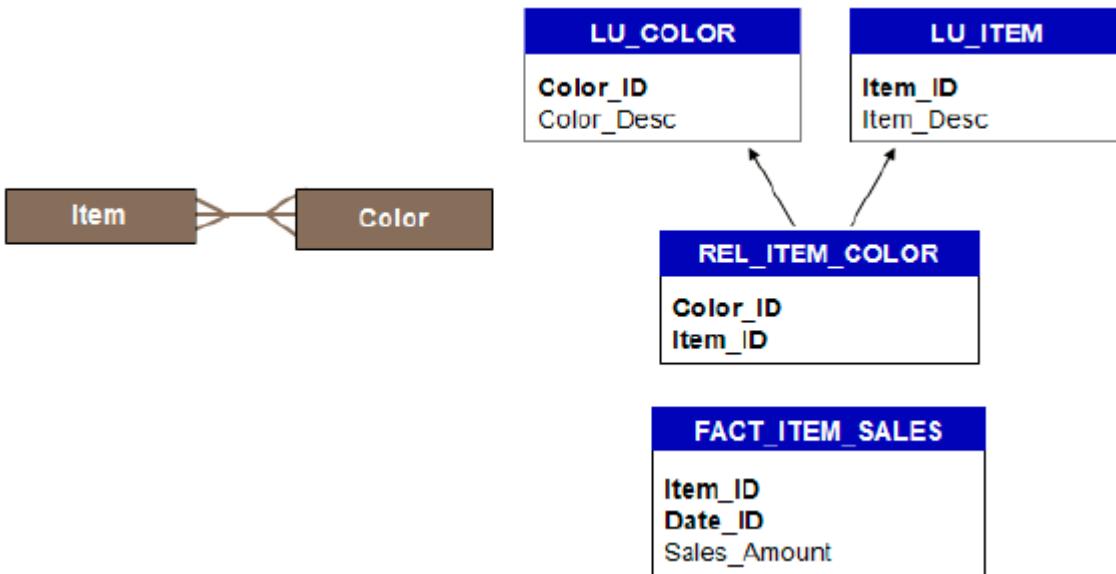
- A table that relates the attributes, identifying all the possible combinations of elements
- A fact table with columns that enable you to accurately join to both the parent and child attributes

Multiple Counting

Lost analytical capability is not the only issue that can arise when dealing with many-to-many relationships. You can also experience problems with multiple counting in certain scenarios. If you try to aggregate data to the level of the parent attribute in the many-to-many relationship (or any attribute level above the parent), multiple counting can occur when the relationship exists in a distinct relationship table but not in the fact table.

The following illustration shows the Color and Item scenario with the original table structure, where only the Item attribute is present in the fact table:

Color and Item—Original Table Structure



To understand how multiple counting can occur, consider the following sample data:

Color and Item—Sample Data

LU_COLOR		LU_ITEM		FACT_ITEM_SALES		
Color_ID	Color_Desc	Item_ID	Item_Desc	Date_ID	Item_ID	Sales_Amount
1	Red	1	Hat	1/3/2012	1 (Hat)	5
2	Blue	2	Dress	2/5/2012	2 (Dress)	25
3	Green	3	Socks	2/8/2012	1 (Hat)	10

REL_ITEM_COLOR	
Color_ID	Item_ID
1 (Red)	1 (Hat)
2 (Blue)	1 (Hat)
3 (Green)	1 (Hat)
1 (Red)	2 (Dress)
2 (Blue)	2 (Dress)
3 (Green)	2 (Dress)
2 (Blue)	3 (Socks)
3 (Green)	3 (Socks)

In this example, there are three items—hats, dresses, and socks. There are also three colors—red, blue, and green. Based on the data in the **REL_ITEM_COLOR** table, hats and dresses are available in all three colors, while socks are available only in blue and green.

Multiple counting occurs when you run any report that requests sales information in conjunction with item colors. The difficulty lies in the fact that color is not in the **FACT_ITEM_SALES** table. Within the fact table, there

is no way to relate the sale of an item to the color of that particular item. You can only determine which items sold, not the colors of the items that sold.

For example, you may want to run a report that aggregates the total sales for hats. Based on the sample data, the result set looks like the following:

Report Result Set with Total Sales for Hats

FACT_ITEM_SALES		
Date_ID	Item_ID	Sales_Amount
1/3/2012	1 (Hat)	5
2/5/2012	2 (Dress)	25
2/8/2012	1 (Hat)	10
3/3/2012	3 (Socks)	2
3/8/2012	2 (Dress)	25
5/2/2012	1 (Hat)	5
6/9/2012	1 (Hat)	10
7/1/2012	3 (Socks)	2
8/9/2012	1 (Hat)	5

→

Resulting Report		
Report details		
Report Filter: Item = Hat		
Metrics	Sales	
Item		
Hat		\$35

The total sales for hats in the report matches the actual data in the FACT_ITEM_SALES table. The SQL Engine can correctly aggregate the total sales for hats based on the information in the fact table. The SQL for this report looks like the following:

```

select a11.[Item_ID] AS Item_ID,
       max(a12.[Item_Desc]) AS Item_Desc,
       sum(a11.[Sales_Amount]) AS WJXBFS1
  from [FACT_ITEM_SALES] a11,
       [LU_ITEM] a12
 where a11.[Item_ID] = a12.[Item_ID]
   and a11.[Item_ID] in (1)
 group by a11.[Item_ID]

```

Notice that only the LU_ITEM and FACT_ITEM_SALES tables appear in the FROM clause. Because the query only asks for hat sales, not hat sales in specific colors, there is no need to include the REL_ITEM_COLOR table in the query.

However, what if you want to know the colors of items that sold? You may want to run a report that aggregates the total sales for all red items. Based on the sample data, the result set looks like the following:

Report Result Set with Total Sales for Red Items

FACT_ITEM_SALES

Date_ID	Item_ID	Sales_Amount
1/3/2012	1 (Hat)	5
2/5/2012	2 (Dress)	25
2/8/2012	1 (Hat)	10
3/3/2012	3 (Socks)	2
3/8/2012	2 (Dress)	25
5/2/2012	1 (Hat)	5
6/9/2012	1 (Hat)	10
7/1/2012	3 (Socks)	2
8/9/2012	1 (Hat)	5

Resulting Report



The resulting report interface consists of two main sections: 'Report details' and a table.

Report details:
Report Filter:
Color = Red

Metrics Sales

Color	Metrics	Sales
Red		\$85

There is no way to accurately determine the total sales for all red items since the Color attribute is not represented in the FACT_ITEM_SALES table. However, the report shows that you sold \$85 of red items. This amount is the total sales for all hats and dresses because those are the two items that are available in red. This number could be correct if all the hats and dresses that were sold were red, but it is more likely incorrect. However, given the available data, this total is what you obtain when the SQL Engine aggregates the item sales data from the Item level to the Color level.

The SQL for this report looks like the following:

```
select a12.[Color_ID] AS Color_ID,
max(a13.[Color_Desc]) AS Color_Desc,
sum(a11.[Sales_Amount]) AS WJXBFS1
from [FACT_ITEM_SALES] a11,
[REL_ITEM_COLOR] a12,
[LU_COLOR] a13
where a11.[Item_ID] = a12.[Item_ID] and
a12.[Color_ID] = a13.[Color_ID]
and a12.[Color_ID] in (1)
group by a12.[Color_ID]
```

Notice that the REL_ITEM_COLOR table is in the FROM clause along with the LU_COLOR and FACT_ITEM_SALES tables. In the WHERE clause, the SQL Engine joins the FACT_ITEM_SALES and REL_ITEM_COLOR tables using only the Item_ID column since Color_ID is not part of the fact table. However, the SQL Engine has to join the REL_ITEM_COLOR and LU_COLOR tables based on the Color_ID column. There is also a filtering condition in the WHERE clause to retrieve only items whose Color_ID value is 1 (red). As a result, the SQL Engine retrieves the sales for all items that are available in red, even though there is no way of knowing whether the items sold were actually red.

Finally, what if you want to know how much you sold of an item in a particular color? You may want to run a report that aggregates the total sales for all red dresses. Based on the sample data, the result set looks like the following:

Report Result Set with Total Sales for Red Dresses

FACT_ITEM_SALES		
Date_ID	Item_ID	Sales_Amount
1/3/2012	1 (Hat)	5
2/5/2012	2 (Dress)	25
2/8/2012	1 (Hat)	10
3/3/2012	3 (Socks)	2
3/8/2012	2 (Dress)	25
5/2/2012	1 (Hat)	5
6/9/2012	1 (Hat)	10
7/1/2012	3 (Socks)	2
8/9/2012	1 (Hat)	5

Resulting Report

Report details		
Report Filter: (Color = Red) And (Item = Dress)		
Metrics		
Color	Item	Sales
Red	Dress	\$50

Again, there is no way to accurately determine the total sales for all red dresses since the Color attribute is not represented in the FACT_ITEM_SALES table. Nevertheless, the report shows that you sold \$50 of red dresses. This amount is the total sales for all dresses. This number could be correct if all the dresses that were sold were red, but it is more likely incorrect. Then again, given the available data, this total is what you obtain when the SQL Engine aggregates the dress sales data from the Item level to the Color level. The SQL for this report looks like the following:

```

select a12.[Color_ID] AS Color_ID,
max(a13.[Color_Desc]) AS Color_Desc,
a11.[Item_ID] AS Item_ID,
max(a14.[Item_Desc]) AS Item_Desc,
sum(a11.[Sales_Amount]) AS WJXBFS1
from [FACT_ITEM_SALES] a11,
[REL_ITEM_COLOR] a12,
[LU_COLOR] a13,
[LU_ITEM] a14
where a11.[Item_ID] = a12.[Item_ID] and
a12.[Color_ID] = a13.[Color_ID] and
a11.[Item_ID] = a14.[Item_ID]
and (a12.[Color_ID] in (1)

```

and a11.[Item_ID] in (2))

group by a12.[Color_ID],

a11.[Item_ID]

Notice that the REL_ITEM_COLOR table is in the FROM clause, along with the LU_COLOR, LU_ITEM, and FACT_ITEM_SALES tables. In the WHERE clause, the SQL Engine joins the FACT_ITEM_SALES and REL_ITEM_COLOR tables using only the Item_ID column since Color_ID is not part of the fact table. However, the SQL Engine has to join the REL_ITEM_COLOR and LU_COLOR tables based on the Color_ID column.

There are also two filtering conditions in the WHERE clause—one to retrieve only items whose Color_ID value is 1 (red) and one to retrieve only items whose Item_ID value is 2 (dress). As a result, the SQL Engine retrieves the sales for all dresses that are sold, even though there is no way of knowing whether the dresses sold were actually red.

In both of the last two report examples, multiple counting occurs precisely because the Color_ID column is not present in the FACT_ITEM_SALES table. Just as you need a relationship table and a fact table that enables you to join to both the parent and child attributes to ensure that you do not lose analytical capability, these same requirements are also necessary to prevent multiple counting when you aggregate data above the level of the child attribute.

Resolving Many-to-Many Relationships

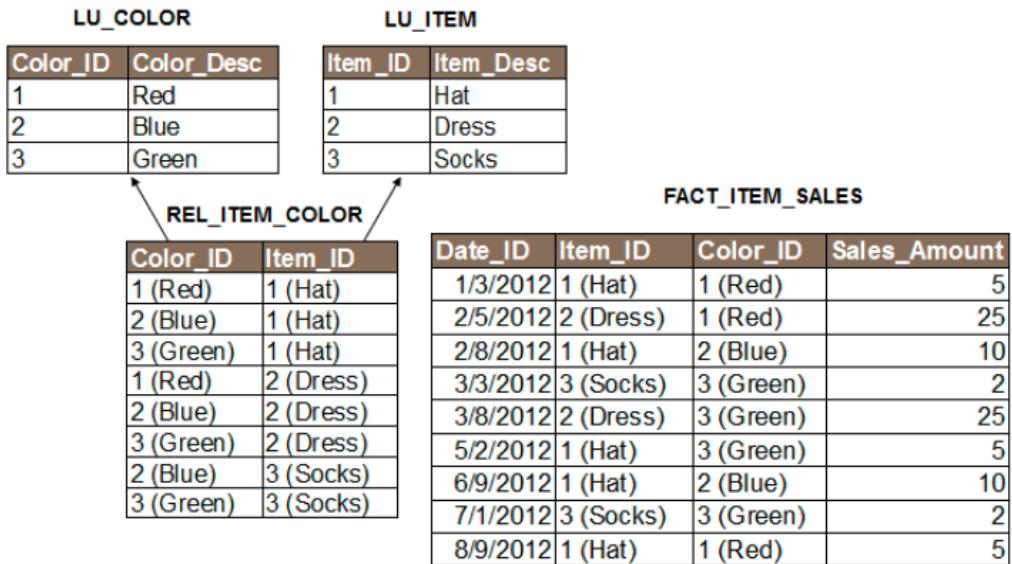
Working with attributes that have many-to-many relationships is more complex than other types of direct relationships. You need to design the logical data model and data warehouse schema to account for the difficulties they can pose.

You can implement many-to-many relationships using one of the following methods:

- Creating a separate relationship table
- Creating a compound child attribute
- Creating a hidden common compound child attribute
- Creating a common child attribute

Each of these methods involves a different design for the logical data model and data warehouse schema. Though, each method retains the two fundamental components that you need to adequately support many-to-many relationships:

- A table that defines the direct attribute relationship
- A fact table structure that enables you to accurately join fact data to both the parent and child attributes

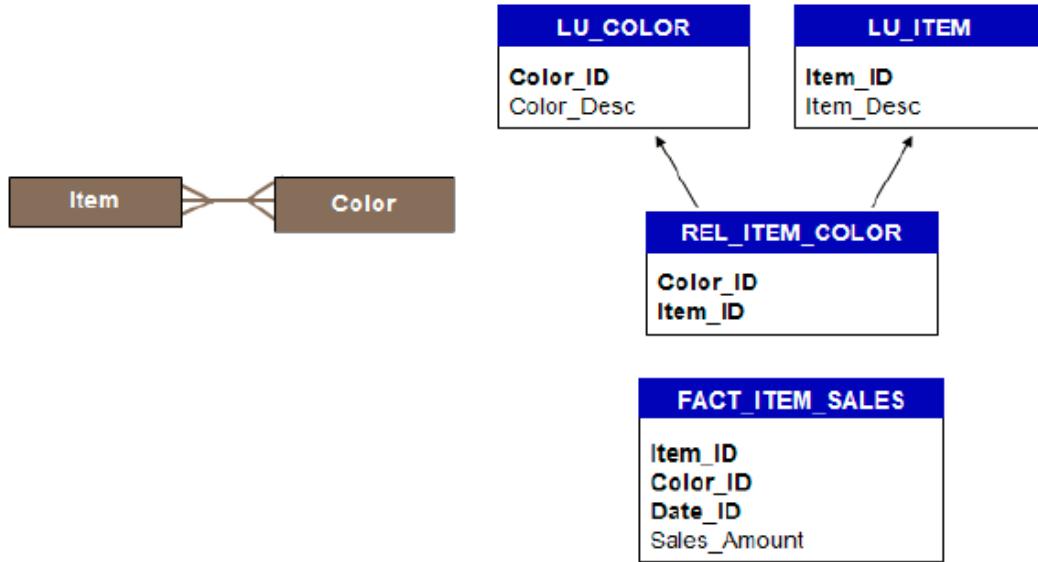


Creating a Separate Relationship Table

Creating a separate relationship table is the most straightforward way in which to effectively manage many-to-many relationships. This method keeps the many-to-many relationship intact and structures the data warehouse schema to resolve any analysis or aggregation issues. You have already seen this method in the examples provided earlier.

You retain the many-to-many relationship between the two attributes and create a separate relationship table that stores all of the possible attribute element combinations. You also add both the parent and child attribute IDs to any fact tables that contain facts you want to analyze with respect to this attribute relationship. The following illustration shows the structure of the logical data model and schema for the Color and Item scenario if you use this method:

Logical Data Model and Schema—Separate Relationship Table



You map the Color attribute to the Color_ID column in the LU_COLOR, REL_ITEM_COLOR, and FACT_ITEM_SALES tables. You map the Item attribute to the Item_ID column in the LU_ITEM, REL_ITEM_COLOR, and FACT_ITEM_SALES tables.

You can then configure a many-to-many relationship between the Color and Item attributes using REL_ITEM_COLOR as the relationship table. In this relationship, Color is the parent attribute, and Item is its child. If you want to view a list of all the possible item and color combinations, you can run a report that just contains the Item and Color attributes.

Report Result Set with All Item and Color Combinations

REL_ITEM_COLOR

Color_ID	Item_ID
1 (Red)	1 (Hat)
2 (Blue)	1 (Hat)
3 (Green)	1 (Hat)
1 (Red)	2 (Dress)
2 (Blue)	2 (Dress)
3 (Green)	2 (Dress)
2 (Blue)	3 (Socks)
3 (Green)	3 (Socks)

Resulting Report



Item	Color	
Hat	Red	
	Blue	
	Green	
Dress	Red	
	Blue	
	Green	
Socks	Blue	
	Green	

This report contains only the two attributes with no metrics. When you have both attributes on a report without a metric, the SQL Engine uses the relationship table to retrieve the list of all item and color combinations. The SQL for this report looks like the following:

```

select a12.[Item_ID] AS Item_ID,
a13.[Item_Desc] AS Item_Desc,
a11.[Color_ID] AS Color_ID,
a11.[Color_Desc] AS Color_Desc
from [LU_COLOR] a11,
[REL_ITEM_COLOR] a12,
[LU_ITEM] a13
where a11.[Color_ID] = a12.[Color_ID] and
a12.[Item_ID] = a13.[Item_ID]

```

Notice that the FROM clause contains the REL_ITEM_COLOR tables along with the lookup tables for each attribute. If you want to view the item and color combinations that have sold, you can run a report that contains the Item and Color attributes along with a Sales metric.

Report Result Set with Sales for Item and Color Combinations

FACT_ITEM_SALES

Date_ID	Item_ID	Color_ID	Sales_Amount
1/3/2012	1 (Hat)	1 (Red)	5
2/5/2012	2 (Dress)	1 (Red)	25
2/8/2012	1 (Hat)	2 (Blue)	10
3/3/2012	3 (Socks)	3 (Green)	2
3/8/2012	2 (Dress)	3 (Green)	25
5/2/2012	1 (Hat)	3 (Green)	5
6/9/2012	1 (Hat)	2 (Blue)	10
7/1/2012	3 (Socks)	3 (Green)	2
8/9/2012	1 (Hat)	1 (Red)	5

Resulting Report

Item	Color	Metrics	Sales
Hat	Red	\$10	
	Blue	\$20	
	Green	\$5	
Total		\$35	
Dress	Red	\$25	
	Green	\$25	
	Total		\$50
Socks	Green	\$4	
	Total		\$4
	Total		\$89

This report contains the two attributes and a metric. When you have both attributes on a report with a metric, the SQL Engine uses the fact table to retrieve the list of all item and color combinations that have sold. The SQL for this report looks like the following:

```

select a11.[Item_ID] AS Item_ID,
       max(a13.[Item_Desc]) AS Item_Desc,
       a11.[Color_ID] AS Color_ID,
       max(a12.[Color_Desc]) AS Color_Desc,
       sum(a11.[Sales_Amount]) AS WJXBFS1
  from [FACT_ITEM_SALES] a11,
       [LU_COLOR] a12,
       [LU_ITEM] a13
 where a11.[Color_ID] = a12.[Color_ID] and
       a11.[Item_ID] = a13.[Item_ID]
 group by a11.[Item_ID],
       a11.[Color_ID]
  
```

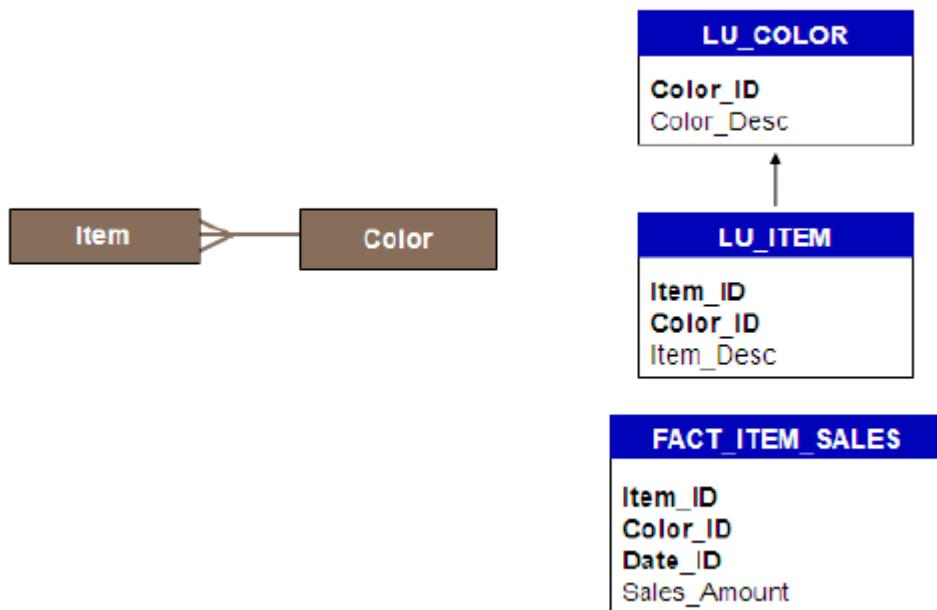
Notice that the FROM clause does not use the relationship table. Instead, it contains the FACT_ITEM_SALES table along with the lookup tables for each attribute.

Creating a Compound Child Attribute

Another method for resolving a many-to-many relationship is to eliminate the many-to-many relationship by converting it into a compound attribute relationship. This method changes both the logical data model and the data warehouse schema.

You create a compound key for the lower-level attribute in the relationship, which eliminates the need for a separate relationship table. This compound key consists of the IDs of the child and parent attributes. With the compound key in place, the attributes essentially have a one-to-many relationship, so you can relate them using the lookup table of the child attribute. You also add the compound attribute ID, which includes both the parent and child attribute IDs, to any fact tables that contain facts you want to analyze with respect to this attribute relationship. The following illustration shows the structure of the logical data model and schema for the Color and Item scenario if you use this method:

Logical Data Model and Schema—Compound Child Attribute



Since Item is the lower-level attribute, you create a compound key for this attribute that consists of the Item_ID and Color_ID columns.

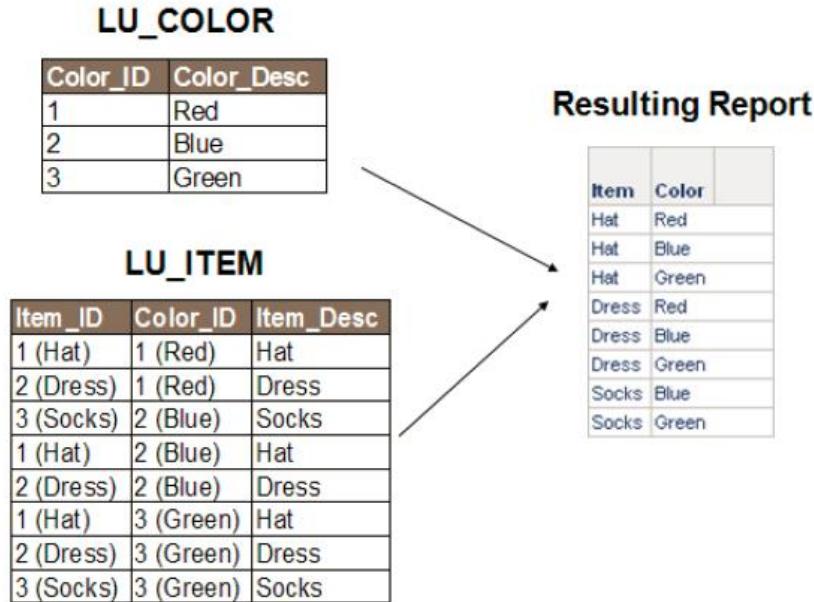
You map the Color attribute to the COLOR_ID column in the the LU_COLOR table. You map the Item attribute to the combination of the Item_ID and Color_ID columns in the LU_ITEM and FACT_ITEM_SALES tables.

You can then configure a one-to-many relationship between the Color and Item attributes using the LU_ITEM table to relate the two attributes. In this relationship, Color is the parent attribute, and Item is its child.

While this method eliminates the many-to-many relationship and the need for a separate relationship table, it also has disadvantages. You have to store an extra column in the LU_ITEM table. Also, joins between the LU_ITEM and FACT_ITEM_SALES tables are more complex because you have to join on a compound key rather than a simple key.

Because this method uses a compound attribute, it also presents some challenges in the reporting environment. With Item defined as a compound attribute, you lose the ability to view items independent of their colors. Each item and color combination is treated as a separate record. Therefore, you cannot merge row headers for items with the same description but different colors. If you run a report that lists all item and color combinations, the result set looks like the following:

Report Result Set with All Item and Color Combinations



The description for each item is repeated for every color in which it is available since you cannot merge the row headers. The SQL for this report looks like the following:

```

select a11.[Item_ID] AS Item_ID,
a11.[Color_ID] AS Color_ID,
a11.[Item_Desc] AS Item_Desc,
a11.[Color_ID] AS Color_ID0,
a12.[Color_Desc] AS Color_Desc
from [LU_ITEM] a11,
[LU_COLOR] a12
where a11.[Color_ID] = a12.[Color_ID]

```

In the WHERE clause, notice that the SQL Engine relates the two attributes using the Color_ID columns in the LU_ITEM and LU_COLOR tables. A separate relationship table is no longer necessary.

Having Item defined as a compound attribute also prevents you from aggregating all the sales data for an item, regardless of color. If you run a report that shows the sales for all item and color combinations that have sold and include subtotals and grand total on the report, the result set looks like the following:

Report Result Set with Sales for Item and Color Combinations

FACT_ITEM_SALES

Date_ID	Item_ID	Color_ID	Sales_Amount
1/3/2012	1 (Hat)	1 (Red)	5
2/5/2012	2 (Dress)	1 (Red)	25
2/8/2012	1 (Hat)	2 (Blue)	10
3/3/2012	3 (Socks)	3 (Green)	2
3/8/2012	2 (Dress)	3 (Green)	25
5/2/2012	1 (Hat)	3 (Green)	5
6/9/2012	1 (Hat)	2 (Blue)	10
7/1/2012	3 (Socks)	3 (Green)	2
8/9/2012	1 (Hat)	1 (Red)	5

Resulting Report

Item	Color	Metrics	Sales
Hat	Red		\$10
	Total		\$10
Hat	Blue		\$20
	Total		\$20
Hat	Green		\$5
	Total		\$5
Dress	Red		\$25
	Total		\$25
Dress	Green		\$25
	Total		\$25
Socks	Green		\$4
	Total		\$4
Total			\$89

The grand total on the report correctly displays the total sales for all items. However, because each item and color combination is treated as a separate item, the item-level subtotal cannot show the total sales for each item for all colors that sold. Instead, you have a separate subtotal for every item and color combination. The SQL for this report looks like the following:

```

select a11.[Item_ID] AS Item_ID,
a11.[Color_ID] AS Color_ID,
max(a12.[Item_Desc]) AS Item_Desc,
a11.[Color_ID] AS Color_ID0,
max(a13.[Color_Desc]) AS Color_Desc,
sum(a11.[Sales_Amount]) AS WJXBFS1
from [FACT_ITEM_SALES] a11,
[LU_ITEM] a12,
[LU_COLOR] a13
where a11.[Color_ID] = a12.[Color_ID] and
a11.[Item_ID] = a12.[Item_ID] and
a11.[Color_ID] = a13.[Color_ID]
group by a11.[Item_ID],
a11.[Color_ID],
a11.[Color_ID]

```

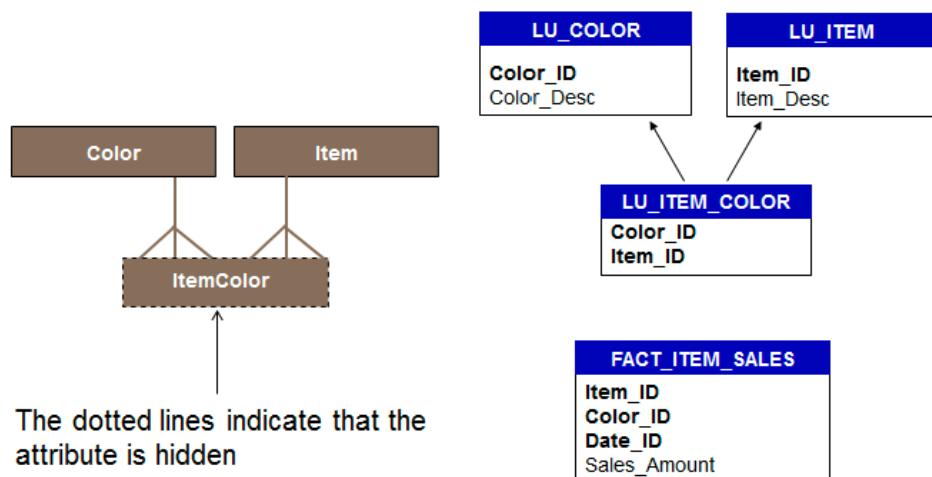
In the WHERE clause, notice that the SQL Engine joins the sales data to the item and color data using the Item_ID and Color_ID columns in the FACT_ITEM_SALES, LU_ITEM, and LU_COLOR tables. Also, the Color_ID column is listed twice in the SELECT and GROUP BY clauses—once as part of the Item attribute and once for the Color attribute.

Creating a Hidden Common Compound Child Attribute

Another alternative for resolving a many-to-many relationship is to remove the direct relationship between the two attributes and create a common child attribute that relates them. This child attribute is hidden because it will not be used to display on a report but only to ensure that the report results are accurate.

With this method, you create a new child attribute that is a concatenation of the original attributes. This attribute is a child of each of the original attributes. It has a one-to-many relationship to both parent attributes. You still need to include the ID of both the parent attributes in any fact tables that contain facts you want to analyze with respect to these attribute relationships. The following illustration shows the structure of the logical data model and schema for the Color and Item scenario if you use this method:

Logical Data Model and Schema—Hidden Common Compound Child Attribute



The ItemColor attribute represents all the item and color combinations. It is a compound attribute and it relates to both the Color and Item attributes in the LU_ITEM_COLOR table. The FACT_ITEM_SALES table is still keyed with the Item and Color attribute IDs.

The ItemColor attribute is a real attribute that exists in the logical data model. However, because it does not carry any logical meaning or users in the reporting environment, you should not include it in the user hierarchy for users to view or browse. Its primary purpose is to relate the Color and Item attributes so that you can consolidate the join path to the lookup tables. Because this attribute is used only in the background, you should make it a hidden attribute.

You can then configure a one-to-many relationship between the Color and ItemColor and Item and ItemColor attributes, using the LU_ITEM_COLOR table to relate the ItemColor attribute to both Color and Item. In this relationship, Color and Item are the parent attributes, and ItemColor is a child of both attributes. The Color and Item attributes are no longer directly related to each other.

This method produces the same result sets as if you created a separate relationship table, but it uses the ItemColor attribute to translate each item and color combination into a compound value. You can use the ItemColor attribute to join the lookup tables without using it on the report template. For example, if you want to view a list of all the possible item and color combinations, you can run a report that just contains the Item and Color attributes:

Report Result Set with All Item and Color Combinations			
FACT_ITEM_SALES			
Date_ID	Item_ID	Color_ID	Sales_Amount
1/3/2012	1 (Hat)	1 (Red)	5
2/5/2012	2 (Dress)	1 (Red)	25
2/8/2012	1 (Hat)	2 (Blue)	10
3/3/2012	3 (Socks)	3 (Green)	2
3/8/2012	2 (Dress)	3 (Green)	25
5/2/2012	1 (Hat)	3 (Green)	5
6/9/2012	1 (Hat)	2 (Blue)	10
7/1/2012	3 (Socks)	3 (Green)	2
8/9/2012	1 (Hat)	1 (Red)	5

→

Item	Color	Metrics	Sales
		Red	\$10
Hat	Blue		\$20
	Green		\$5
	Total		\$35
Dress	Red		\$25
	Green		\$25
	Total		\$50
Socks	Green		\$4
	Total		\$4
	Total		\$89

This report correctly displays the various item and color combinations. Although the ItemColor attribute is not on the template, the SQL Engine uses it to join the item and color data. The SQL for this report looks like the following:

```
select distinct a12.[Item_ID] AS Item_ID,
a13.[Item_Desc] AS Item_Desc,
a11.[Color_ID] AS Color_ID,
a11.[Color_Desc] AS Color_Desc
from [LU_COLOR] a11,
[LU_ITEM_COLOR] a12,
[LU_ITEM] a13
where a11.[Color_ID] = a12.[Color_ID] and
a12.[Item_ID] = a13.[Item_ID]
```

Notice that the FROM clause includes the LU_ITEM_COLOR table. In the WHERE clause, the SQL Engine uses the Color_ID and Item_ID columns to join the LU_ITEM_COLOR table to the LU_COLOR and LU_ITEM tables.

If you want to view the item and color combinations that have sold, you can run a report that contains the Item and Color attributes along with a Sales metric:

Report Result Set with Sales for Item and Color Combinations

The diagram illustrates the transformation of a fact table into a report. On the left, the **FACT_ITEM_SALES** table is shown with columns: Date_ID, Item_ID, Color_ID, and Sales_Amount. The data consists of 10 rows. An arrow points from this table to the right, where the **Resulting Report** is displayed as a table with columns: Item, Color, Metrics, and Sales. The report shows sales for three items (Hat, Dress, Socks) across three colors (Red, Blue, Green), including totals for each item and overall totals.

FACT_ITEM_SALES				→	Resulting Report		
Date_ID	Item_ID	Color_ID	Sales_Amount		Item	Color	Metrics
1/3/2012	1 (Hat)	1 (Red)	5		Hat	Red	\$10
2/5/2012	2 (Dress)	1 (Red)	25			Blue	\$20
2/8/2012	1 (Hat)	2 (Blue)	10			Green	\$5
3/3/2012	3 (Socks)	3 (Green)	2			Total	\$35
3/8/2012	2 (Dress)	3 (Green)	25		Dress	Red	\$25
5/2/2012	1 (Hat)	3 (Green)	5			Green	\$25
6/9/2012	1 (Hat)	2 (Blue)	10			Total	\$50
7/1/2012	3 (Socks)	3 (Green)	2		Socks	Green	\$4
8/9/2012	1 (Hat)	1 (Red)	5			Total	\$4

The SQL for this report looks like the following:

```

select a11.[Item_ID] AS Item_ID,
       max(a13.[Item_Desc]) AS Item_Desc,
       a11.[Color_ID] AS Color_ID,
       max(a12.[Color_Desc]) AS Color_Desc,
       sum(a11.[Sales_Amount]) AS WJXBFS1
  from [FACT_ITEM_SALES] a11,
       [LU_COLOR] a12,
       [LU_ITEM] a13
 where a11.[Color_ID] = a12.[Color_ID] and
       a11.[Item_ID] = a13.[Item_ID]
 group by a11.[Item_ID],
       a11.[Color_ID]
  
```

Notice that the `FROM` clause does not use the `LU_ITEM_COLOR` table. Instead, it contains the `FACT_ITEM_SALES` table along with the lookup tables for each attribute.

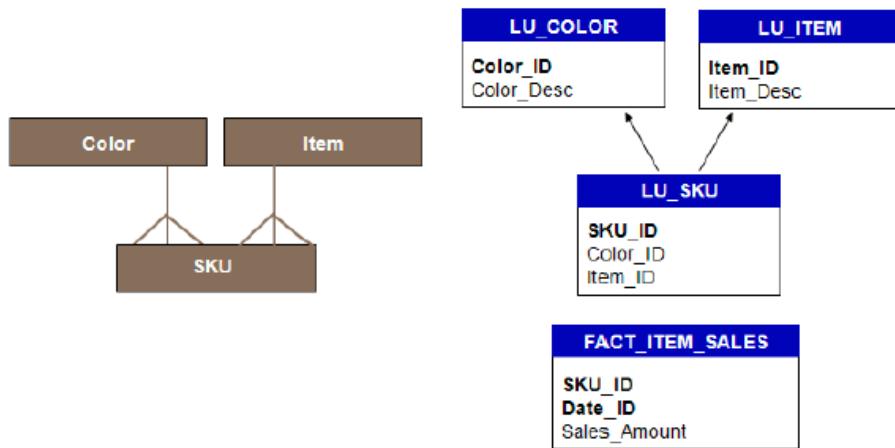
Creating a Common Child Attribute

The fourth option for resolving a many-to-many relationship is to implement a variation of the previous method. It too eliminates the many-to-many relationship and the need for a separate relationship table. It uses only a simple attribute rather than a compound attribute. It requires only one attribute column in fact tables rather than two attribute columns.

With this method, you create a new attribute that is a concatenation of the original attributes. This attribute is a child of each of the original attributes. It has a one-to-many relationship to both parent attributes. You include the ID of this new attribute in any fact tables that contain facts you want to analyze with respect to

these attribute relationships. The following illustration shows the structure of the logical data model and schema for the Color and Item scenario if you use this method:

Logical Data Model and Schema—Common Child Attribute



The SKU attribute represents all the item and color combinations. It has its own ID column, but it relates to both the Color and Item attributes in the LU_SKU table. You key the FACT_ITEM_SALES table using the SKU attribute ID rather than the Item and Color attribute IDs.

You map the SKU attribute to the SKU_ID columns in the LU_SKU and FACT_ITEM_SALES tables. You map the Item attribute to the Item_ID column in the LU_SKU and LU_ITEM tables, and you map the Color attribute to the Color_ID column in the LU_SKU and LU_COLOR tables.

You can then configure a one-to-many relationship between the Color and SKU and Item and SKU attributes, using the LU_SKU table to relate the SKU attribute to both Color and Item. In this relationship, Color and Item are the parent attributes, and SKU is a child of both attributes. The Color and Item attributes are no longer directly related to each other.

This method produces the same result sets as if you created a separate relationship table, but it uses the SKU attribute to translate each item and color combination into a single value. You can use the SKU attribute to join fact table and lookup table data without using it on the report template. For example, if you want to view a list of all the possible item and color combinations, you can run a report that just contains the Item and Color attributes:

Report Result Set with All Item and Color Combinations

LU_SKU

Item_ID	Color_ID	SKU_ID
1 (Hat)	1 (Red)	1
2 (Dress)	1 (Red)	2
3 (Socks)	2 (Blue)	3
1 (Hat)	2 (Blue)	4
2 (Dress)	2 (Blue)	5
1 (Hat)	3 (Green)	6
2 (Dress)	3 (Green)	7
3 (Socks)	3 (Green)	8

Resulting Report

Item	Color
Hat	Red
	Blue
	Green
Dress	Red
	Blue
	Green
Socks	Blue
	Green

This report correctly displays the various item and color combinations. Although the SKU attribute is not on the template, the SQL Engine uses it to join the item and color data. The SQL for this report looks like the following:

```
select distinct a12.[Item_ID] AS Item_ID,  
a13.[Item_Desc] AS Item_Desc,  
a11.[Color_ID] AS Color_ID,  
a11.[Color_Desc] AS Color_Desc  
from [LU_COLOR] a11,  
[LU_SKU] a12,  
[LU_ITEM] a13  
where a11.[Color_ID] = a12.[Color_ID] and  
a12.[Item_ID] = a13.[Item_ID]
```

Notice that the FROM clause includes the LU_SKU table. In the WHERE clause, the SQL Engine uses the Color_ID and Item_ID columns to join the LU_SKU table to the LU_COLOR and LU_ITEM tables. If you want to view the item and color combinations that have sold, you can run a report that contains the Item and Color attributes along with a Sales metric:

Report Result Set with Sales for Item and Color Combinations

FACT_ITEM_SALES

Date_ID	SKU_ID	Sales_Amount
1/3/2012	1 (Red Hat)	5
2/5/2012	2 (Red Dress)	25
2/8/2012	4 (Blue Hat)	10
3/3/2012	8 (Green Socks)	2
3/8/2012	7 (Green Dress)	25
5/2/2012	6 (Green Hat)	5
6/9/2012	4 (Blue Hat)	10
7/1/2012	8 (Green Socks)	2
8/9/2012	1 (Red Hat)	5

Resulting Report

Item	Color	Metrics	Sales
Hat	Red		\$10
	Blue		\$20
	Green		\$5
	Total		\$35
Dress	Red		\$25
	Green		\$25
	Total		\$50
Socks	Green		\$4
	Total		\$4
		Total	\$89

This report correctly displays the sales for each item and color combination as well as the appropriate subtotals and the grand total. Although the SKU attribute is not on the template, the SQL Engine uses it to join the sales data to the item and color data. The SQL for this report looks like the following:

```

select a12.[Item_ID] AS Item_ID,
       max(a14.[Item_Desc]) AS Item_Desc,
       a12.[Color_ID] AS Color_ID,
       max(a13.[Color_Desc]) AS Color_Desc,
       sum(a11.[Sales_Amount]) AS WJXBFS1
  from [FACT_ITEM_SALES] a11,
       [LU_SKU] a12,
       [LU_COLOR] a13,
       [LU_ITEM] a14
 where a11.[SKU_ID] = a12.[SKU_ID] and
       a12.[Color_ID] = a13.[Color_ID] and
       a12.[Item_ID] = a14.[Item_ID]
 group by a12.[Item_ID],
       a12.[Color_ID]
  
```

This method provides several advantages. Using a simple key in the fact table is more efficient, and it reduces the number of columns in the table, which can be significant in a large fact table.

The only disadvantage of this method lies in the changes you have to make to the data warehouse schema to support creating the new attribute. You have to create the lookup table for the new attribute and key fact tables using the ID of the new attribute, which can add complexity to the ETL process.

Attribute roles

Attribute roles refer to an attribute's ability to play multiple roles using a single definition in the business model. This occurs when a column in a single lookup table is used to define more than one attribute.

For example, your project might include Destination Airport and Origin Airport attributes, both of which are defined using the AIRPORT_ID and AIRPORT_DESC columns in the LU_AIRPORT lookup table. Although these attributes share the same forms, the AIRPORT_FACT table contains a distinct column for each role - ORIGIN_AIRPORT_ID and DESTINATION_AIRPORT_ID.

If an analyst creates a report that displays the number of flights that originated from MIA and arrived at LGA, an empty result set is returned. This is because the SQL Engine attempts to find an AIRPORT_ID that is MIA and LGA at the same time, which is not possible.

To account for a single column that is used to create multiple attributes, you must create attribute roles using one of the following methods:

- Create an explicit table alias definition
- Turn on automatic attribute role recognition in cases where attribute roles employ the same expression.

Create an explicit table alias definition

A table alias creates a distinct copy of a logical table in your schema. For example, you can create a LU_AIRPORT_ALIAS table alias based on the LU_AIRPORT table. You can then map Destination Airport attribute to one of these logical tables, and map Origin Airport to the other.

When you create a report that includes both attributes, the SQL Engine will create a query that returns data because it will leverage the two logical lookup tables.

Turn on automatic attribute role recognition

Automatic attribute role recognition enables MicroStrategy to automatically detect when multiple attributes use the same column in a lookup table. To leverage this function, enable the Engine Attribute Role Options VLDB property at the database instance level. When the option is enabled, a table alias is automatically created for a detected attribute role.

Although this option simplifies the attribute role creation process, it limits the amount of control you have over the schema. This is because the automatically created aliases cannot be edited.

Attribute roles are intended to work with lookup tables and should not be applied to a fact table. If you have a rare use case that requires attribute roles to be created based on a fact table, create an explicit table alias.

Automatic attribute role recognition does not detect attributes that share a child attribute.

Exercise 13.1: Configure attribute roles

Attribute roles enable you to use a single column in a lookup table to create multiple attributes in your project. To leverage this feature, you must either create an explicit table alias or turn on automatic attribute role recognition.

In this exercise, in the **Attribute Roles Project - AARR Project**, you will create and run reports in the project without having any solution in place to enable attribute roles. You will then configure the project for attribute roles using automatic attribute role recognition. Finally, you will disable automatic attribute role recognition for the project and use explicit table aliasing to support attribute roles.

Attribute Roles Project - AARR project information

In the data model for this project, Customer City and Store City are role attributes that both point to the same lookup table. Because you are focusing on these attributes, only the bottom two levels of each hierarchy in the data model exist in the project. The Customer hierarchy is displayed in the following diagram:

Customer Hierarchy

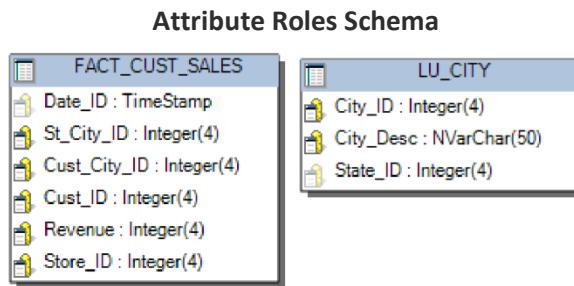


The Store hierarchy looks is displayed in the following diagram:

Store Hierarchy



The schema for this project includes the following two tables:



Both the Customer City and Store City attributes map to the ID and description columns in the lu_city table. The Customer City attribute also maps to the Cust_City_ID column in the FACT_CUST_SALES table, and the Store City attribute maps to the St_City_ID column in this table. For these exercises, since you do not need to query store and customer information, separate lookup tables for the Store and Customer attributes do not exist. These attributes are mapped only to the Store_ID and Cust_ID columns in the FACT_CUST_SALES table.

Create and run a report with role attributes

1. In MicroStrategy Developer, in the **MicroStrategy Analytics Modules Project Source**, open the **Attribute Roles Project - AARR**.

*If you were logged out, you can log in to **MicroStrategy Analytics Modules Project Source** using the following credentials:*

- login id: administrator
- password: (none)

2. In the Public Objects\Reports folder, create the following report:

Report View: 'Local Template'		Switch to:	
Store City	Customer City	Metrics	Revenue

3. Access the Store City attribute from the Store hierarchy and the Customer City attribute from the Customer hierarchy.
4. The Revenue metric is located in the Public Objects\Metrics\Attribute Roles folder.
5. Run the report. The result set resembles the following:

Store City	Customer City	Metrics	Revenue
Los Angeles	Los Angeles		\$199
San Diego	San Diego		\$221
San Francisco	San Francisco		\$200
Sacramento	Sacramento		\$12
Seattle	Seattle		\$182
Boise	Boise		\$375
Billings	Billings		\$85
Great Falls	Great Falls		\$431
Cheyenne	Cheyenne		\$91
Herndon	Herndon		\$286
Fairfax	Fairfax		\$628
Nokesville	Nokesville		\$189
Scarsborough	Scarsborough		\$172
Portland	Portland		\$72
Colorado Springs	Colorado Springs		\$111
Monument	Monument		\$456
Dallas	Dallas		\$36
San Antonio	San Antonio		\$168
Austin	Austin		\$23

The result set is incomplete because both the Customer City and Store City attributes are on the report. The report displays only records where the store city and customer city happen to be the same, and excludes all other records.

6. Switch to **SQL View** for the report. The following SQL is displayed:

```
select a11.Cust_City_ID City_ID,
       max(a12.City_Desc) City_Desc,
       a11.St_City_ID City_ID0,
       max(a12.City_Desc) City_Desc0,
       sum(a11.Revenue) WJXBFS1
  from "FACT_CUST_SALES"      a11
  join "LU_CITY"             a12
    on (a11.Cust_City_ID = a12.City_ID)
   and (a11.St_City_ID = a12.City_ID)
 group by a11.Cust_City_ID,
          a11.St_City_ID
```

The report requires the ID and description for both attributes. To correctly obtain the required information, it needs to access the LU_CITY table twice. Because these two attributes are not configured as role attributes, the table is aliased only once in the FROM clause.

7. Switch to **Grid View**.
8. Save the report in the Reports folder as **Revenue by Store City and Customer City**. Close the report.

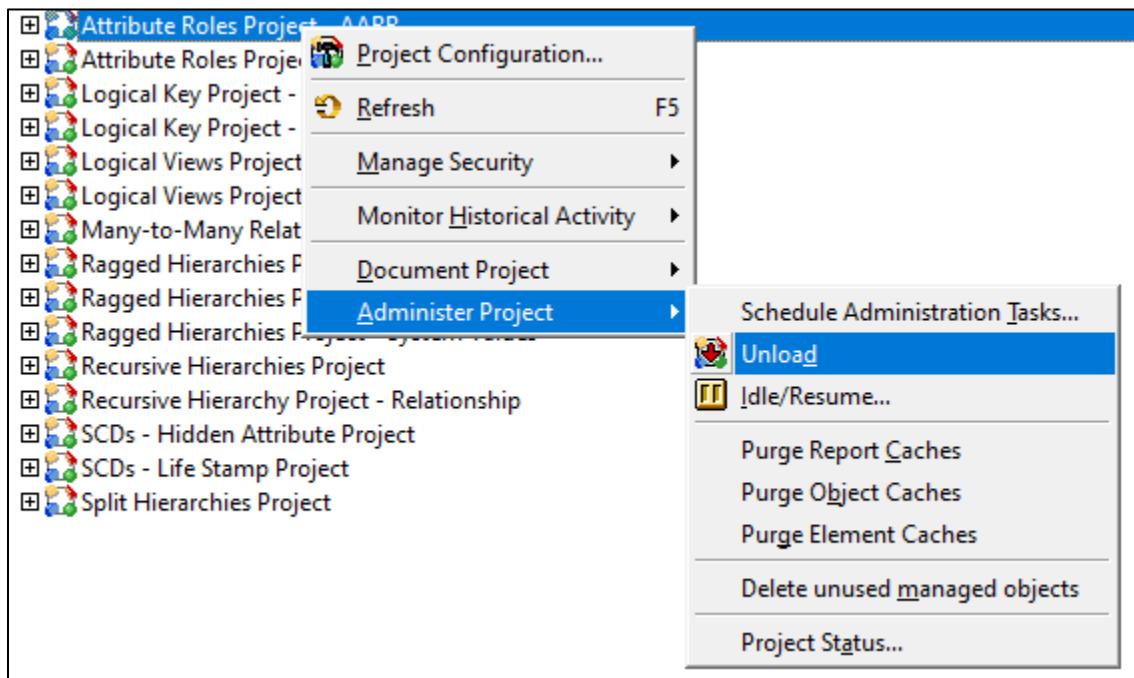
Resolve role attributes using automatic attribute role recognition

1. In the MicroStrategy on **MicroStrategy Analytics Modules Project Source**, expand **Administration**, then **Configuration Managers**, then click **Database Instances**.

2. In the Object Viewer, right-click the **Advanced Data Warehousing Warehouse** database instance and select **VLDB Properties**.
 3. In the VLDB Properties window, from the **Tools** menu, select **Show Advanced Settings** if it is not already enabled.
 4. From the VLDB Settings list, expand the **Query Optimizations** folder and click **Engine Attribute Role Options**.
 5. Clear the **Use default inherited value - (Default Settings)** check box.
 6. Click **Enable Engine Attribute Role Feature**. Then click **Save and Close**.
 7. In the message window regarding restarting Intelligence Server, click **OK**.
-

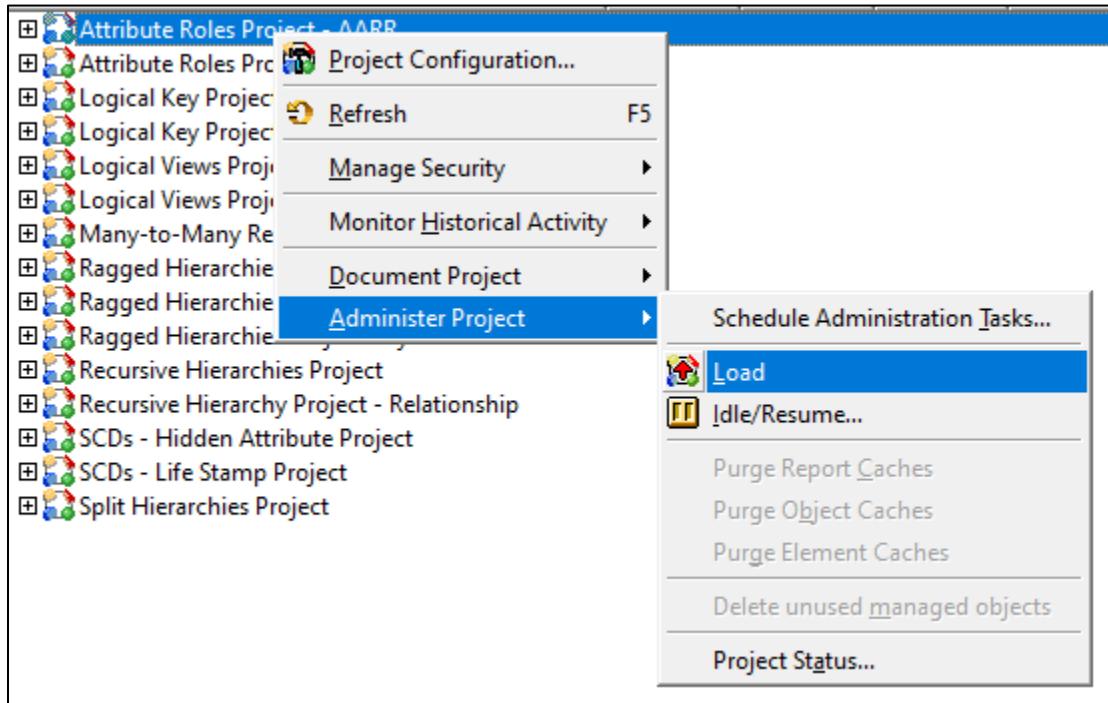
Unload and load the project for the changes to take effect

8. In the MicroStrategy on **MicroStrategy Analytics Modules Project Source**, expand **Administration**, then **System Administration**, then click **Projects**.
9. Right Click on **Attribute Roles Project – AARR** , point to **Administer Project**, and click **Unload**.



10. Wait for few seconds until the project disappears from the folder list.

11. To reload the project, Right Click on **Attribute Roles Project – AARR** , point to **Administer Project**, and click **load**



Run the Revenue by Store City and Customer City report

12. Open the **Attribute Roles Project - AARR**.
13. In the Public Objects\Reports folder, double-click the **Revenue by Store City and Customer City** report to run it. The result set should resemble the following:

Store City	Customer City	Metrics	Revenue
Los Angeles	Los Angeles	\$199	
San Diego	Los Angeles	\$601	
	San Diego	\$221	
San Francisco	San Francisco	\$200	
	Sacramento	\$578	
Sacramento	Sacramento	\$12	
Seattle	Seattle	\$182	
Tacoma	Seattle	\$129	
Boise	Boise	\$375	
Billing	Billing	\$85	
Great Falls	Great Falls	\$431	
Cheyenne	Cheyenne	\$91	
	Herndon	\$286	
Herndon	Fairfax	\$17	
	Noxville	\$101	
Fairfax	Herndon	\$11	
	Fairfax	\$628	
Noxville	Noxville	\$189	
Scarsborough	Scarsborough	\$172	
Portland	Scarsborough	\$73	
	Portland	\$72	
	Colorado Springs	\$111	
Colorado Springs	Woodland Park	\$82	
	Black Forest	\$190	
Denver	Monument	\$168	
Monument	Monument	\$456	
Dallas	Dallas	\$36	
	Austin	\$45	
San Antonio	San Antonio	\$168	
Austin	San Antonio	\$17	
	Austin	\$23	

Notice that the result set now correctly returns all records in the fact table; not just the records where Store City and Customer City are the same.

If your results did not change, delete the report cache:

- a. In the MicroStrategy on **MicroStrategy Analytics Modules Project Source**, expand Administration, and then expand **System Monitors**.
- b. Expand **Caches** and click **Reports**.
- c. In the Cache Monitor Options window, select **Attribute Roles Project - AARR** and click **OK**.
- d. Select the cache and press **Delete** on your keyboard.

14. Switch to **SQL View** for the report. You should see the following SQL:

```

select a11.Cust_City_ID City_ID,
       max(a12.City_Desc) City_Desc,
       a11.St_City_ID City_ID0,
       max(a13.City_Desc) City_Desc0,
       sum(a11.Revenue) WJXBFS1
  from `FACT_CUST_SALES`      a11
       join `LU_CITY`        a12
         on (a11.Cust_City_ID = a12.City_ID)
       join `LU_CITY`        a13
         on (a11.St_City_ID = a13.City_ID)
 group by a11.Cust_City_ID,
          a11.St_City_ID
    
```

Notice that the LU_CITY table is aliased twice in the FROM clause. The query can retrieve all records from the fact table; not just those where the Store City and Customer City are the same.

15. Close the report. You do not need to save the report.

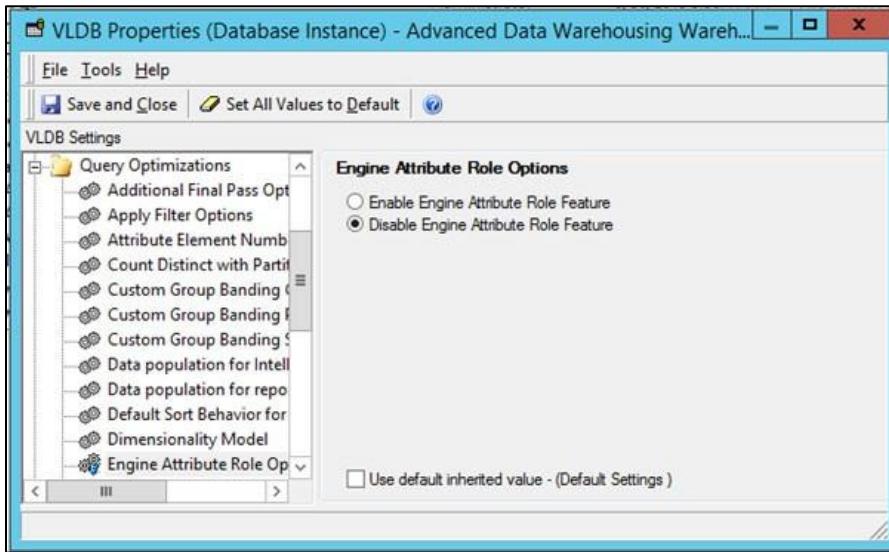
Now that you have seen how to support role attributes using automatic attribute role recognition, you will disable automatic attribute role recognition and create an explicit table alias to support role attributes.

Resolve role attributes using explicit table aliasing

16. In Developer, in the MicroStrategy on MicroStrategy Analytics Modules Project Source, expand **Administration**, then expand **Configuration Managers**, and click **Database Instances**.



17. In the Object Viewer, right-click the **Advanced Data Warehousing Warehouse** database instance and select **VLDB Properties**.
18. On the Tools menu, select **Show Advanced Settings**, if not selected already.
19. In the VLDB Properties window, in the VLDB Settings list, expand the **Query Optimizations** folder.
20. In the Query Optimizations folder, click **Engine Attribute Role Options**.
21. Deselect **Use default inherited value (Default Settings)** and select **Disable Engine Attribute Role Feature**.



22. Click **Save and Close**. In the message window regarding restarting Intelligence Server, click **OK**.
23. Disconnect from the MicroStrategy Analytics Modules Project Source.
24. Reconnect to the MicroStrategy Analytics Modules Project Source

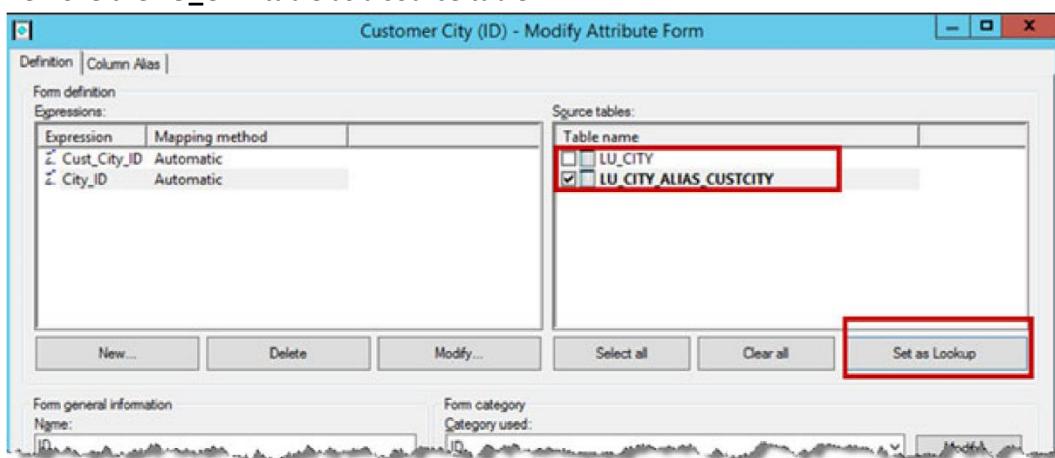
You must disconnect and reconnect to the project source for the change in the Engine Attribute Role VLDB property to take effect.

Create an explicit table alias

25. Open the **Attribute Roles Project - AARR**.
26. In the Schema Objects\Tables, right-click the **LU_CITY** table and select **Create Table Alias**. The **LU_CITY(1)** table is created.
27. Right-click the **LU_CITY (1)** table alias and select **Rename**.
28. Rename the table alias **LU_CITY_ALIAS_CUSTCITY**.

Map the Customer City attribute

29. In the Schema Objects\Attributes folder, open the **Customer City** attribute.
30. Modify the **ID** form to map the **City_ID** form expression to the **LU_CITY_ALIAS_CUSTCITY** table.
31. Select the **LU_CITY_ALIAS_CUSTCITY** table and click **Set as Lookup** to set the table as the primary lookup table.
32. Remove the **LU_CITY** table as a source table.



33. Click **OK**.
34. Modify the **DESC** form to map the **City_Desc** form expression to the **LU_CITY_ALIAS_CUSTCITY** table.
35. Select the **LU_CITY_ALIAS_CUSTCITY** table and click **Set as Lookup** to set the table as the primary lookup table.
36. Remove the **LU_CITY** table as a source table.
37. Click **OK**, then save and close the **Customer City** attribute.

Map the Store City attribute

38. Open the **Store City** attribute.
 39. For the ID form, ensure that the **LU_CITY_ALIAS_CUSTCITY** table is not selected as a source table for the City_ID form expression.
 40. Save and close the **Store City** attribute, then update the project schema.
-

Delete the report cache

41. In the MicroStrategy on MicroStrategy Analytics Modules Project Source, expand Administration, and then expand System Monitors.
 42. Expand Caches and click Reports.
 43. In the Cache Monitor Options window, select Attribute Roles Project - AARR and click OK.
 44. Select the cache and press Delete on your keyboard.
-

Re-run the report and view updated result set

45. Run the **Revenue by Store City and Customer City** report again. The result set should look like the following:

Store City	Customer City	Metrics	Revenue
Los Angeles	Los Angeles		\$199
San Diego	Los Angeles		\$601
	San Diego		\$221
San Francisco	San Francisco		\$200
	Sacramento		\$578
Sacramento	Sacramento		\$12
Seattle	Seattle		\$182
Tacoma	Seattle		\$129
Boise	Boise		\$375
Billings	Billings		\$85
Great Falls	Great Falls		\$431
Cheyenne	Cheyenne		\$91
	Herndon		\$286
Herndon	Fairfax		\$17
	Noakesville		\$101
Fairfax	Herndon		\$11
	Fairfax		\$628
Noakesville	Noakesville		\$189
Scarborough	Scarborough		\$172
Portland	Scarborough		\$73
	Portland		\$72
Colorado Springs	Colorado Springs		\$111
Colorado Springs	Woodland Park		\$82
	Black Forest		\$190
Denver	Monument		\$168
Monument	Monument		\$456
Dallas	Dallas		\$36
	Austin		\$45
San Antonio	San Antonio		\$168
Austin	San Antonio		\$17
	Austin		\$23

Notice that the result set again correctly returns all of the records in the fact table; not just the records where the store city and customer city are the same.

46. Switch to **SQL View** for the report. You should see the following SQL:

```
select a11.Cust_City_ID City_ID,
       max(a12.City_Desc) City_Desc,
       a11.St_City_ID City_ID0,
       max(a13.City_Desc) City_Desc0,
       sum(a11.Revenue) WJXBFS1
  from 'FACT_CUST_SALES' a11
       join 'LU_CITY' a12
         on (a11.Cust_City_ID = a12.City_ID)
       join 'LU_CITY' a13
         on (a11.St_City_ID = a13.City_ID)
 group by a11.Cust_City_ID,
          a11.St_City_ID
```

Notice that the LU_CITY table is aliased twice in the FROM clause. The query can retrieve the information for both Store City and Customer City.

47. Close the report. You do not need to save the report.

CHAPTER 14: HIERARCHIES

Ragged Hierarchy

A ragged hierarchy is one in which the organizational structure varies such that the depth of the hierarchy is not uniform. In other words, for every child attribute element, there does not always exist a corresponding parent attribute element. Instead, the child attribute element may have a direct relationship only with a grandparent attribute element.

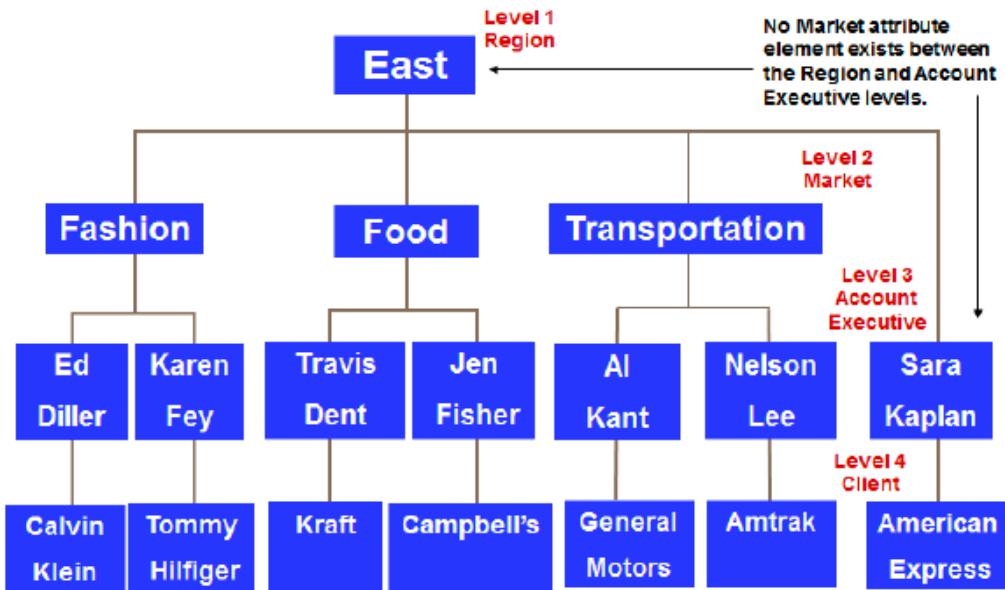
For example, an advertising company has its sales organization represented as follows:

Logical Model for Sales Hierarchy



In this model, the company is divided into regions, which are then split into markets by advertising segments. Each market has dedicated account executives who are responsible for specific clients. However, that general structure may not hold true for all clients. For example, you could have some clients that do not directly correspond to a market. Therefore, account executives for these clients report directly to the region level. A look at some of the actual data reveals a ragged structure to the hierarchy:

Ragged Structure of Sales Data



There are four levels of data in the warehouse for the Sales hierarchy. They map to the logical data model as follows:

- Level 1 maps to the Region attribute.
- Level 2 maps to the Market attribute.
- Level 3 maps to the Account Executive attribute.
- Level 4 maps to the Client attribute.

In the second level of data, notice that there are only three markets—Fashion, Food, and Transportation. One of the elements that ties directly into the East region is an account executive, Sara Kaplan. The physical data at the second level is ragged. The Market attribute is not represented between the Region and Account Executive levels for all of the data in the warehouse. In the case of Sara Kaplan, she is responsible for a single client that is not associated with a particular market, so she falls directly under the East region in the sales structure rather than reporting through a market. For this particular attribute element, the Market attribute carries no meaning, making the hierarchy ragged.

Attributes from ragged hierarchies are problematic when you place them in reports. For example, if you want to create a report that displays all of the attributes in a ragged hierarchy, the missing values can cause issues since data does not exist uniformly at every level to populate each cell in the report.

Ragged hierarchies also pose a challenge when drilling. If a report contains an attribute from a ragged hierarchy and you need to drill to other levels in the hierarchy, values may not exist for every row of data in the original report.

Denormalizing the schema can prevent aggregation issues and resolve specific drill paths. However, even in a denormalized schema, drills or other queries that join through the lookup table in which the gaps exist will continue to pose a challenge as some data will be “left out” of result sets because of the gaps.

You can resolve these issues with ragged hierarchies in one of three ways:

- Use left outer join so that data can be aggregated from skipped level
- Model the attribute relationships so that skipped levels are eliminated
- Populate the gaps for an attribute with values from its parent or child attribute or with system-generated values

Using Left Outer Join

One method for resolving ragged hierarchies is to use left outer join of the lookup table to the fact table. For example, account executives, Sara Kaplan and Dave Williams are not assigned to markets. If you run a report that shows market and account executive information along with the revenue generated by each account executive, the default result set looks like the following:

Result Set for Market and Account Executive with a Metric

Market	Account Executive	Revenue
Food	Jen Fisher	\$5,000
Electronics	Eve Smith	\$6,000
Magazines	Mark Price	\$80,000

Because Sara Kaplan and Dave Williams are not assigned to markets, the null values that exist in the database table cause them to be excluded from the result set.

The SQL for this report looks like the following:

```

select a13.[Market_ID] AS Market_ID,
       max(a14.[Market_Desc]) AS Market_Desc,
       a12.[Acct_Exec_ID] AS Acct_Exec_ID,
       max(a13.[Acct_Exec_Name]) AS Acct_Exec_Name,
       sum(a11.[Revenue]) AS WJXBFS1
  from ((([FACT_CLIENT_SALES] a11
inner join [LU_CLIENT] a12
    on (a11.[Client_ID] = a12.[Client_ID]))
inner join [LU_ACCT_EXEC] a13
    on(a12.[Acct_Exec_ID] = a13.[Acct_Exec_ID]))
inner join [LU_MARKET] a14
    on(a13.[Market_ID] = a14.[Market_ID]))
 group by a13.[Market_ID],
          a12.[Acct_Exec_ID]

```

To address this issue, you first change the join VLDB property of the Market attribute and all of its parents attributes so that data can be aggregated from kipped levels. Second for the affected report, you modify the join VLDB property at the report level. Once you have made both these changes, you can roll up data from the lowest level to higher levels in the hierarchy.

Changing VLDB Properties

To change the VLDB properties, perform the following steps:

1. Edit the Market attribute and on the Tools menu, select VLDB Properties.
2. Expand the Joins folder and select **Preserve** all final pass result elements.
3. Clear the Use default inherited value - (Default Settings) check box.
4. Click Preserve all elements of final pass result table with respect to lookup table but not relationship table.
5. Click Save and Close in the attribute editor. 6 Update the schema.
6. Edit the report and on the Data menu, select VLDB Properties.
7. Expand the Joins folder and select **Preserve** all final pass result elements. 9 Clear the Use default inherited value - (Default Settings) check box.
8. 10 Click Do not listen to per report level setting, preserve elements of the final pass according to the setting at the attribute level. If this choice is selected at attribute level, it will be treated as preserve common elements (i.e. choice 1).
9. Controlling the join setting at the report level ensures that changing the attribute does not affect the entire project. This adds flexibility to your reporting environment because you can choose to have some reports that left join on the attribute while others do not.
10. Save the report.

Now if you run the report, the result set looks like the following:

Result Obtained by Left Outer Join

Market	Account Executive	Revenue
	Sara Kaplan	\$750,000
	Dave Williams	\$250,000
Food	Jen Fisher	\$5,000
Electronics	Eve Smith	\$6,000
Magazines	Mark Price	\$80,000

The SQL for this report looks like the following:

```
select a13.[Market_ID] AS Market_ID,  
max(a14.[Market_Desc]) AS Market_Desc,  
a12.[Acct_Exec_ID] AS Acct_Exec_ID,  
max(a13.[Acct_Exec_Name]) AS Acct_Exec_Name,  
sum(a11.[Revenue]) AS WJXBFS1  
from(((FACT_CLIENT_SALES] a11  
inner join [LU_CLIENT] a12  
on (a11.[Client_ID] = a12.[Client_ID]))  
inner join [LU_ACCT_EXEC] a13  
on a12.[Acct_Exec_ID] = a13.[Acct_Exec_ID]))  
left join [LU_MARKET]a14  
on (a13.[Market_ID] = a14.[Market_ID]))  
group by a13.[Market_ID],  
a12.[Acct_Exec_ID]
```

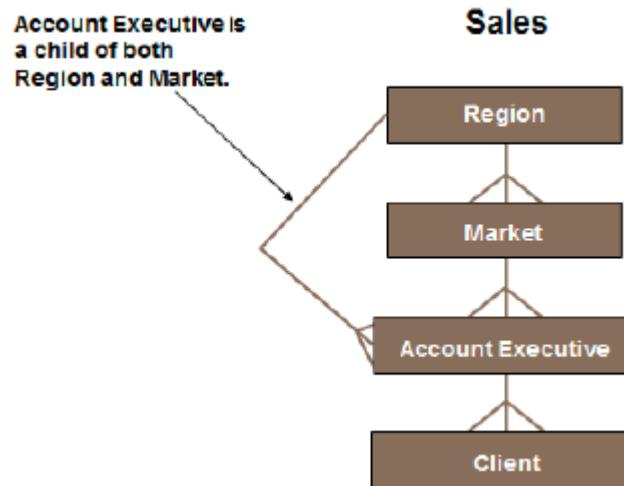
Because of changing the join VLDB property at the attribute and report levels, the report now contains the information for account executives who are not assigned to markets.

Although modifying the VLDB property resolves the issue of displaying data at the Market level, it does not resolve the issue of displaying data at the parent level. For example a report showing Region, Market, Account Executive, and Revenue will result in account executives who are not assigned to markets still not displaying on the report. Therefore, this option is not a complete solution to resolving issues with ragged hierarchies.

Revising the Data Model

Another method for resolving ragged hierarchies is to revise the data model so that gaps do not exist. For example, you could revise the Sales hierarchy to model the attribute relationships as follows:

Revised Logical Model for Sales Hierarchy



Changing the data model to directly relate the Region and Account Executive attributes also means modifying the underlying structure of the LU_ACCT_EXEC table. You need to add the ID column for Region to the LU_ACCT_EXEC table to map the relationship between the two attributes:

Modified Lookup Table for Account Executive

The Region_ID column is added to the LU_ACCT_EXEC table to directly relate regions to account executives.

LU_ACCT_EXEC

Acct_Exec_ID	Acct_Exec_Name	Market_ID	Region_ID
28	Jen Fisher	11 (Food)	1 (East)
32	Sara Kaplan		1 (East)
34	Eve Smith	17 (Electronics)	2 (Central)
37	Dave Williams		3 (West)
40	Mark Price	22 (Magazines)	3 (West)

By adding the Region_ID column to the LU_ACCT_EXEC table and making Account Executive a child of Region, you establish a means of relating account executives directly to their corresponding regions. You can now drill directly from Region to Account Executive without having to join through the lookup table for Market.

Although changing the data model provides a drill path to avoid the gaps in the ragged structure of the hierarchy, it does not resolve issues with displaying data from a ragged hierarchy on a report. If a report contains all levels of the hierarchy, the SQL joins include the lookup table for Market in which the gaps exist. As a result, account executives who are not assigned to markets still do not display on the report. Therefore, this option is not a complete solution to resolving issues with ragged hierarchies.

Populating Null Attribute Values

A better method for resolving ragged hierarchies is to populate the null values with attribute elements of either the child or parent attribute or with system-generated values. Inserting values effectively eliminates gaps in a ragged hierarchy.

For example, the following illustration shows the original data in the LU_ACCT_EXEC table:

Lookup Table for Account Executive with Null Values

Null values exist for the Market attribute for two of the account executives.

LU_ACCT_EXEC

Acct_Exec_ID	Acct_Exec_Name	Market_ID
28	Jen Fisher	11 (Food)
32	Sara Kaplan	
34	Eve Smith	17 (Electronics)
37	Dave Williams	
40	Mark Price	22 (Magazines)

Sara Kaplan and Dave Williams are not assigned to markets, so the market ID for both of them is null. You could run a report with the following template in which all three attribute levels are present:

Template with Attributes from Sales Hierarchy



However, if you run this report, Sara Kaplan and Dave Williams are not included in the result set since their respective market IDs are null:

Report Result with Null Values

Because Sara Kaplan and Dave Williams are not assigned to markets, the null values that exist in the database table cause them to be excluded from the result set.



Region	Market	Account Executive
East	Food	Jen Fisher
Central	Electronics	Eve Smith
West	Magazines	Mark Price

To ensure that all account executives are included in the report display, you can populate the empty values in the LU_ACCT_EXEC table by inserting the values of the parent (Region) or child (Account Executive) attributes into the Market_ID column of the LU_ACCT_EXEC table or by generating your own values to replace the nulls.

If you populate the Market_ID column with the parent attribute values, the LU_ACCT_EXEC table looks like the following:

Lookup Table for Account Executive with Parent Values

The Market ID column is populated with the Region attribute values.

LU_ACCT_EXEC



Acct_Exec_ID	Acct_Exec_Name	Market_ID
28	Jen Fisher	11 (Food)
32	Sara Kaplan	23 (East)
34	Eve Smith	17 (Electronics)
37	Dave Williams	24 (West)
40	Mark Price	22 (Magazines)

Now, if you run the same report, the result set looks like the following:

Report Result with Parent Values

The Region values display under the Market attribute in the report.

Region	Market	Account Executive
East	Food	Jen Fisher
	East	Sara Kaplan
Central	Electronics	Eve Smith
	Magazines	Mark Price
West	West	Dave Williams

Alternately, you could populate the empty cells for the Market_ID column with the values for the Account Executive attribute. Then, the LU_ACCT_EXEC table looks like the following:

Lookup Table for Account Executive with Child Values

The Market ID column is populated with the Account Executive attribute values.

LU_ACCT_EXEC

Acct_Exec_ID	Acct_Exec_Name	Market_ID
28	Jen Fisher	11 (Food)
32	Sara Kaplan	32 (Sara Kaplan)
34	Eve Smith	17 (Electronics)
37	Dave Williams	37 (Dave Williams)
40	Mark Price	22 (Magazines)

Now, if you run the same report, the result set looks like the following:

Report Result with Child Values

The Account Executive values display under the Market attribute in the report.

Region	Market	Account Executive
East	Food	Jen Fisher
	Sara Kaplan	Sara Kaplan
Central	Electronics	Eve Smith
	Magazines	Mark Price
West	Dave Williams	Dave Williams

Whether you choose to populate empty cells with the parent or child attribute values completely depends on which action provides the most business value to users as they view reports.

If inserting parent or child attribute values does not make sense in your business environment, you can also populate the empty cells with system-generated IDs that map to descriptions that indicate that a value does not exist.

For example, you could generate market IDs for account executives who are not assigned to a market. In the lookup table for the Market attribute, these IDs map to description columns that indicate that no market is assigned. Then, the LU_ACCT_EXEC table looks like the following:

Lookup Table for Account Executive with Generated Values

The Market ID column is populated with the system-generated values.

LU_ACCT_EXEC		
Acct_Exec_ID	Acct_Exec_Name	Market_ID
28	Jen Fisher	11 (Food)
32	Sara Kaplan	25 (No Assigned Market - East)
34	Eve Smith	17 (Electronics)
37	Dave Williams	26 (No Assigned Market - West)
40	Mark Price	22 (Magazines)

Now, if you run the same report, the result set looks like the following:

Report Result with Generated Values

The system-generated values display under the Market attribute in the report, indicating to users that markets are not assigned to these account executives.



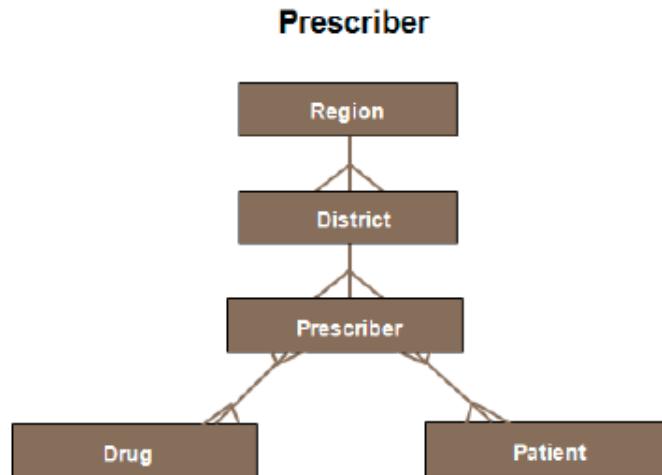
Region	Market	Account Executive
East	Food	Jen Fisher
	No Assigned Market – East	Sara Kaplan
Central	Electronics	Eve Smith
	Magazines	Mark Price
West		Dave Williams
No Assigned Market – West		

Split Hierarchy

A split hierarchy is one in which there is a split in the primary hierarchy such that more than one child attribute exists at some level in the hierarchy. Most hierarchies follow a linear progression from higher-level to lower-level attributes. While characteristic attributes may branch off the primary hierarchy at various points, the primary hierarchy itself generally follows a single path to the lowest-level attribute and any related fact tables.

With split hierarchies, somewhere along the primary hierarchy, a split occurs. For example, a pharmaceutical company has its Prescriber hierarchy organized as follows:

Logical Data Model for Prescriber Hierarchy



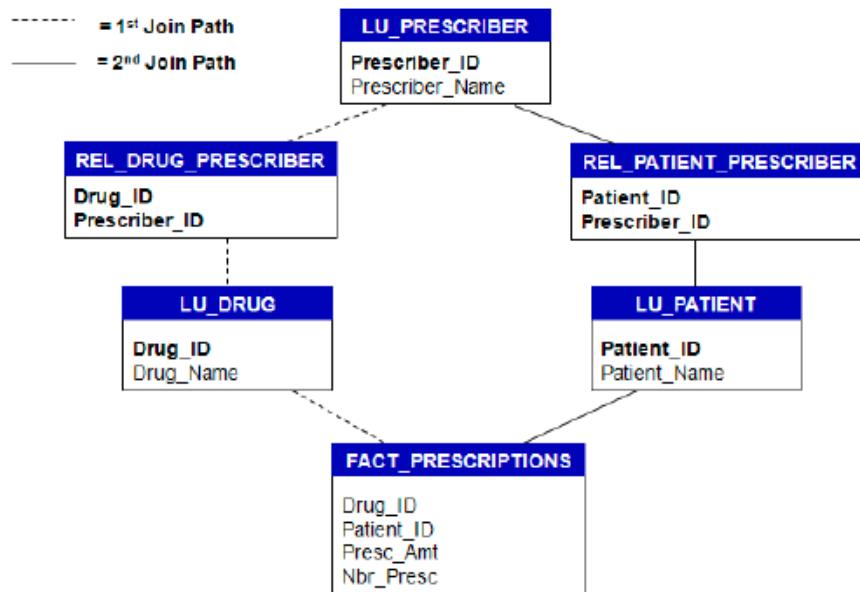
The prescriber relates at the lowest level to both the drug that is being prescribed and the patient to whom the prescription belongs. In this example, the complexity is compounded by the many-to-many relationships between the parent and child attributes. A prescriber can prescribe multiple drugs, and multiple prescribers can prescribe the same drug. A prescriber has multiple patients, and a patient can go to multiple prescribers (doctors) for different ailments.

The problem with a split hierarchy is that it provides two paths that you can use to join to fact tables. The lookup tables (and relationship tables in the case of many-to-many relationships) for each parent-child attribute form separate, distinct join paths. You can use either path to join to fact tables for metrics that are contained in a report. Nonetheless, the SQL Engine optimizes the path to fact tables, so it is forced to make a choice.

For split hierarchies in which there are one-to-one or one-to-many relationships between the parent and children, the split results in the SQL Engine consistently choosing one join path over the other, even though that path may not be the most efficient way of joining from the fact tables to the parent attribute for all queries. This same problem also arises when you have split hierarchies in which there are many-to-many relationships. However, the many-to-many relationship further compounds the issue. Because the SQL Engine chooses one join path over the other, the join may not occur through the proper relationship table, which can lead to an inaccurate result set.

For example, the Prescriber hierarchy contains the following tables:

Tables in Prescriber Hierarchy



There are lookup tables for the Prescriber, Drug, and Patient attributes as well as two separate relationship tables—one to map the relationship between Prescriber and Drug and one to map the relationship between Prescriber and Patient. These tables contain the following data:

Table Data

LU_PRESCRIBER		REL_DRUG_PRESCRIBER		REL_PATIENT_PRESCRIBER	
		Drug_ID	Prescriber_ID	Patient_ID	Prescriber_ID
1	Evan Thomas	1 (Zestril)	1 (Evan Thomas)	1 (Cindy Tyler)	1 (Evan Thomas)
2	Frank Hanford	1 (Zestril)	2 (Frank Hanford)	1 (Cindy Tyler)	2 (Frank Hanford)
3	Tacia Hunt	2 (Levaquin)	3 (Tacia Hunt)	2 (Sam Tate)	3 (Tacia Hunt)
		3 (Histussin)	1 (Evan Thomas)	3 (Henry Moore)	1 (Evan Thomas)
		3 (Histussin)	3 (Tacia Hunt)	4 (Tina Drake)	2 (Frank Hanford)
				4 (Tina Drake)	3 (Tacia Hunt)

LU_DRUG		LU_PATIENT	
Drug_ID	Drug_Name	Patient_ID	Patient_Name
1	Zestril	1	Cindy Tyler
2	Levaquin	2	Sam Tate
3	Histussin	3	Henry Moore
		4	Tina Drake

FACT_PRESCRIPTIONS			
Drug_ID	Patient_ID	Presc_Amt	Nbr_Presc
3 (Histussin)	1 (Cindy Tyler)	90	3
2 (Levaquin)	2 (Sam Tate)	120	2
3 (Histussin)	3 (Henry Moore)	45	1
1 (Zestril)	4 (Tina Drake)	80	1
2 (Levaquin)	4 (Tina Drake)	180	3
3 (Histussin)	4 (Tina Drake)	90	2

If you run a report to view the drugs that prescribers have prescribed, the result set looks like the following:

Result Set for Prescriber-Drug Information

Prescriber	Drug
Evan Thomas	Zestril Histussin
Frank Hanford	Zestril
Tricia Hunt	Levaquin Histussin

The result set correctly displays each prescriber along with the drugs they have prescribed. The SQL for this report looks like the following:

```
select a12.[Prescriber_ID] AS Prescriber_ID,
a13.[Prescriber_Name] AS [Prescriber_Name],
a11.[Drug_ID] AS Drug_ID,
a11.[Drug_Name] AS Drug_Name
from [LU_DRUG] a11,
[REL_DRUG_PRESCRIBER] a12,
[LU_PRESCRIBER] a13
where a11.[Drug_ID] = a12.[Drug_ID] and
a12.[Prescriber_ID] = a13.[Prescriber_ID]
```

The result set is correct because it is obtained by querying the REL_DRUG_PRESCRIBER table, which maps the relationships between prescribers and drugs.

You could also run a report to view the patients for each prescriber. The result set looks like the following:

Result Set for Prescriber-Patient Information

Prescriber	Patient
Evan Thomas	Cindy Tyler
Frank Hanford	Henry Moore
Tricia Hunt	Cindy Tyler
	Tina Drake
	Sam Tate
	Tina Drake

The result set correctly displays each prescriber along with the patients for whom they have prescribed drugs.

The SQL for this report looks like the following:

```
select a12.[Prescriber_ID] AS Prescriber_ID,
a13.[Prescriber_Name] AS [Prescriber_Name],
a11.[Patient_ID] AS Patient_ID,
a11.[Patient_Name] AS Patient_Name
```

```

from [LU_PATIENT] a11,
[REL_PATIENT_PRESCRIBER] a12,
[LU_PRESCRIBER] a13
where a11.[Patient_ID] = a12.[Patient_ID] and
a12.[Prescriber_ID] = a13.[Prescriber_ID]

```

The result set is correct because it is obtained by querying the REL_PATIENT_PRESCRIBER table, which maps the relationships between prescribers and patients.

You could also run a report that shows prescriber and patient information along with the amount of prescriptions for each patient. The result set looks like the following:

Result Set for Prescriber-Patient Information with a Metric

Prescriber	Patient	Metrics	Prescription Amount
Evan Thomas	Cindy Tyler		\$90
	Henry Moore		\$45
	Tina Drake		\$170
Frank Hanford	Tina Drake		\$80
	Cindy Tyler		\$90
	Sam Tate		\$120
Tricia Hunt	Henry Moore		\$45
	Tina Drake		\$270

With the Prescription Amount metric as part of the report, this result set does not correctly display the prescriber and patient relationships. Instead of relating patients to prescribers who have prescribed drugs for them, the result set relates patients to any prescriber that prescribes drugs they have taken, regardless of whether they actually obtained their prescription from that particular prescriber.

The SQL for this report looks like the following:

```

select a12.[Prescriber_ID] AS Prescriber_ID, max(a14.[Prescriber_Name]) AS [Prescriber_Name],
a11.[Patient_ID] AS Patient_ID,
max(a13.[Patient_Name]) AS Patient_Name,
sum(a11.[Presc_Amt]) AS WJZBFS1
from [FACT_PRESCRIPTIONS] a11,
[REL_DRUG_PRESCRIBER] a12,
[LU_PATIENT] a13,
[LU_PRESCRIBER] a14
where a11.[Drug_ID] = a12.[Drug_ID] and
a11.[Patient_ID] = a13.[Patient_ID] and

```

```
a12.[Prescriber_ID] = a14.[Prescriber_ID]  
group by a12.[Prescriber_ID],  
a11.[Patient_ID]
```

In this case, the result set is incorrect because the SQL Engine chooses to join from the FACT_PRESCRIPTIONS table to the LU_PRESCRIBER table through the Drug attribute, rather than the Patient attribute. Therefore, it uses the relationship table between Drug and Prescriber to obtain the result set. As a result, the query finds the drugs that each prescriber prescribed and then just joins to each patient who took those drugs, regardless of whether or not they have a relationship with a particular prescriber. To determine relationships between patients and prescribers (the information that you really want to analyze in this report), the query must access the relationship table between Prescriber and Patient. Because the SQL Engine chooses the join path provided by the Drug attribute, the REL_PATIENT_PRESCRIBER table is not included in the query.

A preliminary step to generating SQL is that the SQL Engine has to determine the most efficient join path from fact tables to lookup tables. To do so, the SQL Engine checks the following:

- Logical table size
- Order of attributes in the system hierarchy

MicroStrategy Architect automatically calculates the logical table size and assigns a numeric value to each table relative to the attributes that are contained in the table and their position in their respective hierarchies.

Usually, a smaller logical table size equates to a smaller physical table size. In cases like this one where the SQL Engine finds two join paths to the LU_PRESCRIBER table, it checks the logical size of both the LU_DRUG and LU_PATIENT tables. In this example, the Drug and Patient attributes have the same weight. Therefore, the tables have the same logical size. The SQL Engine cannot differentiate between the two paths based on logical table size.

At this point, if the logical table size is equal, the SQL Engine cannot distinguish which path is the most efficient. Therefore, it simply picks the lookup table based on the order of the attributes (Drug and Patient) in the system hierarchy. Because Drug is first in the system hierarchy (it was created before Patient), the SQL Engine chooses to join through the LU_DRUG table.

Essentially, when you have a split hierarchy such as this one, the SQL Engine has no way of differentiating between the join paths because it creates a situation in which both choices seem to be equally efficient. In actuality, depending on the attributes on the report, sometimes you need to join through the LU_DRUG table and sometimes through the LU_PATIENT table.

The best way to ensure that the SQL Engine always selects the most efficient join path and uses tables that provide the desired result set is to remove the split from the hierarchy. You can resolve split hierarchies by creating joint child relationships.

Creating a Joint Child

The joint child relates each of the original parent and child attributes that are involved in the split. It also provides a single path for joins. In this example, you need to relate Prescriber, Drug, and Patient. To set up the joint child, you need to do the following:

- Create a relationship table that includes the parent attribute and both child attributes.
- Create joint child relationship between the parent and the two children attributes using this relationship table.

Creating the Relationship Table

First, you need to create a relationship table that maps the relationship between the parent attribute and both child attributes. The relationship table looks like the following:

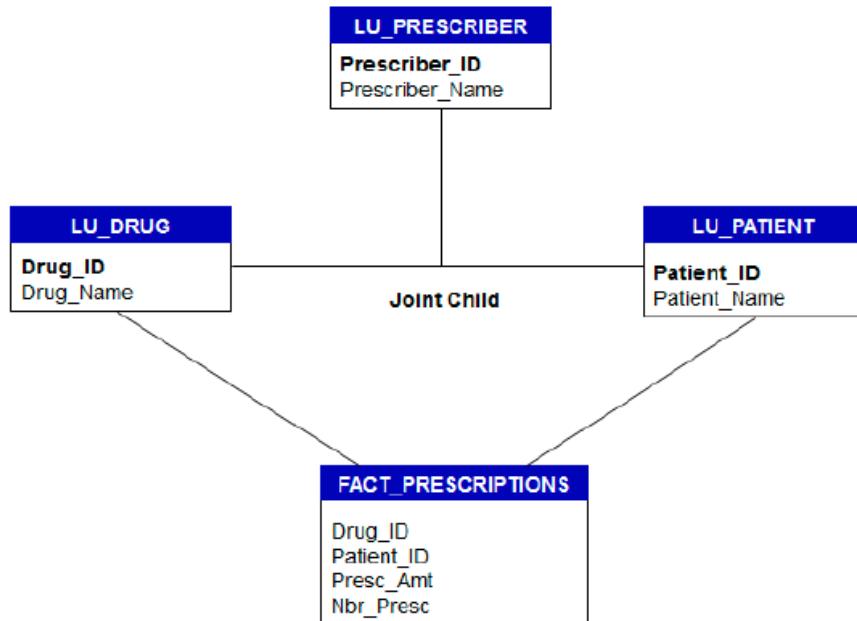
Relationship Table with All Three Attributes

REL_PRESCRIBER_DRUG_PATIENT

Prescriber_ID	Drug_ID	Patient_ID
1 (Evan Thomas)	3 (Histussin)	1 (Cindy Tyler)
1 (Evan Thomas)	3 (Histussin)	3 (Henry Moore)
2 (Frank Hanford)	1 (Zestril)	4 (Tina Drake)
3 (Tacia Hunt)	2 (Levaquin)	2 (Sam Tate)
3 (Tacia Hunt)	2 (Levaquin)	4 (Tina Drake)
3 (Tacia Hunt)	3 (Histussin)	4 (Tina Drake)

The REL_PRESCRIBER_DRUG_PATIENT table provides a means of relating all three attributes to one another. In this way, you can view prescribers in relationship to both the drugs they prescribe and the patients to whom they prescribe them at the same time. The following image shows the modified relationship between the three attributes:

Modified Tables in the Prescriber Hierarchy

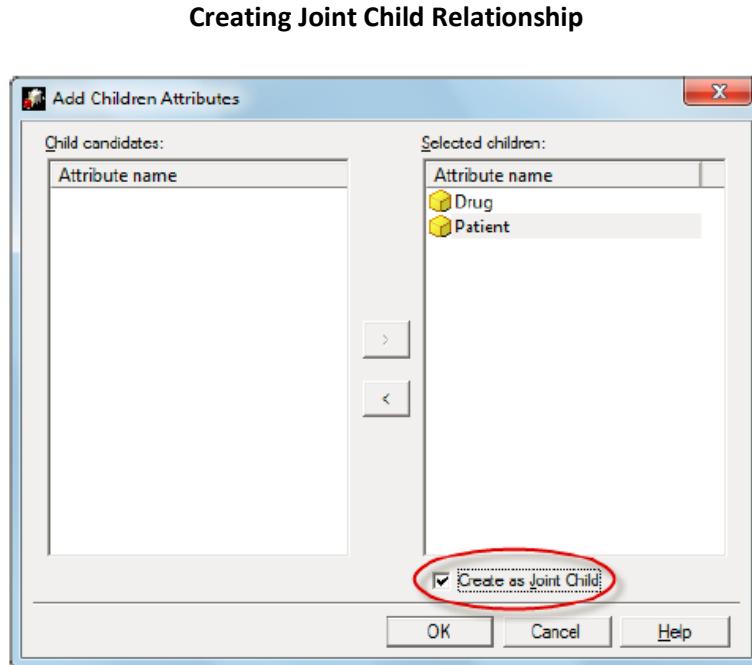


Creating the Joint Child

After creating the relationship table, you need to perform the following steps:

1. Add the REL_PRESCRIBER_DRUG_PATIENT table to the project.
2. Map the ID forms of the Prescriber, Drug and Patient attributes to the respective ID columns in the REL_PRESCRIBER_DRUG_PATIENT table.
3. Unmap the ID forms of the Prescriber, Drug and Patient attributes from the REL_PATIENT_PRESCRIBER and LU_DRUG_PRESCRIBER tables.
4. Remove the REL_PATIENT_PRESCRIBER and
5. LU_DRUG_PRESCRIBER tables from the project.
6. Add Drug and Patient attributes as children of the Prescriber attribute.
7. Make sure to select the joint child check box.

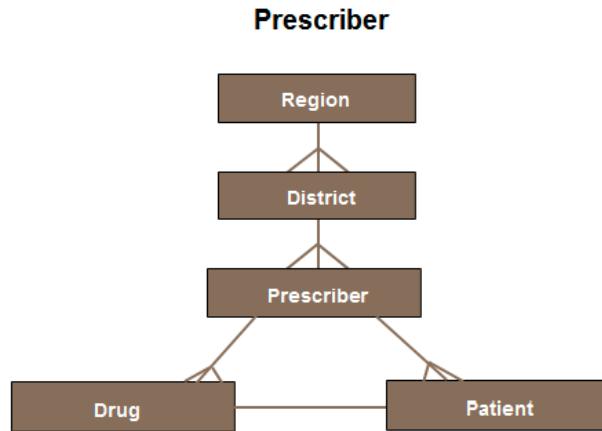
The following image shows the option for setting the joint child relationship:



8. Update the project schema.

In the illustration above, when you make the Patient and Drug attributes joint children of Prescriber, you create a structure where the two separate children attributes involved in the original split are now jointly related to each other through the parent attribute. Essentially, you modify the logical data model to look like the following:

Modified Logical Data Model for Prescriber Hierarchy



Now, if you run the same report to obtain prescriber and patient information along with the prescription amounts for each patient, the result set looks like the following:

Result Set Obtained by Joint Child Relationship

REL_PRESCRIBER_DRUG_PATIENT

Prescriber_ID	Drug_ID	Patient_ID
1 (Evan Thomas)	3 (Histussin)	1 (Cindy Tyler)
1 (Evan Thomas)	3 (Histussin)	3 (Henry Moore)
2 (Frank Hanford)	1 (Zestril)	4 (Tina Drake)
3 (Tacia Hunt)	2 (Levaquin)	2 (Sam Tate)
3 (Tacia Hunt)	2 (Levaquin)	4 (Tina Drake)
3 (Tacia Hunt)	3 (Histussin)	4 (Tina Drake)

FACT_PRESCRIPTIONS

Drug_ID	Patient_ID	Presc_Amt	Nbr_Presc
3 (Histussin)	1 (Cindy Tyler)	90	3
2 (Levaquin)	2 (Sam Tate)	120	2
3 (Histussin)	3 (Henry Moore)	45	1
1 (Zestril)	4 (Tina Drake)	80	1
2 (Levaquin)	4 (Tina Drake)	180	3

→

Prescriber	Patient	Metrics	Prescription Amount
Evan Thomas	Cindy Tyler		\$90
	Henry Moore		\$45
Frank Hanford	Tina Drake		\$80
Tricia Hunt	Sam Tate		\$120
	Tina Drake		\$270

Because of the joint child, the report now contains the correct result set since the information from the FACT_PRESCRIPTIONS table is joined to the prescriber information through the REL_PRESCRIBER_DRUG_PATIENT table.

The SQL looks like the following:

```

select a12.[Prescriber_ID] AS Prescriber_ID,
       max(a14.[Prescriber_Name]) AS
       Prescriber_Name,
  
```

```

a12.[Patient_ID] AS Patient_ID,
max(a13.[Patient_Name]) AS Patient_Name,
sum(a11.[Presc_Amt] AS WJXBFS1
from [FACT_PRESCRIPTIONS] a11,
[REL_PREScriber_DRUG_PATIENT] a12,
[LU_PATIENT] a13,
[LU_PREScriber] a14
where a11.[Drug_ID] = a12.[Drug_ID] and
a11.[Patient_ID] = a12.[Patient_ID]
a11.[Patient_ID] = a13.[Patient_ID] and
a12.[Prescriber_ID] = a14.[Prescriber_ID]
group by a12.[Prescriber_ID],
a12.[Patient_ID]

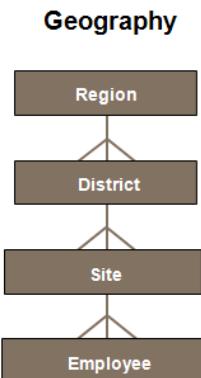
```

Notice that the relationship table appears in the FROM clause. The WHERE clause uses the same relationship table to join to the fact table. As a result, the SQL Engine can now efficiently join to the fact table for either the Drug or Patient attribute and deliver a valid result set that correctly portrays the relationships between prescribers and drugs or prescribers and patients.

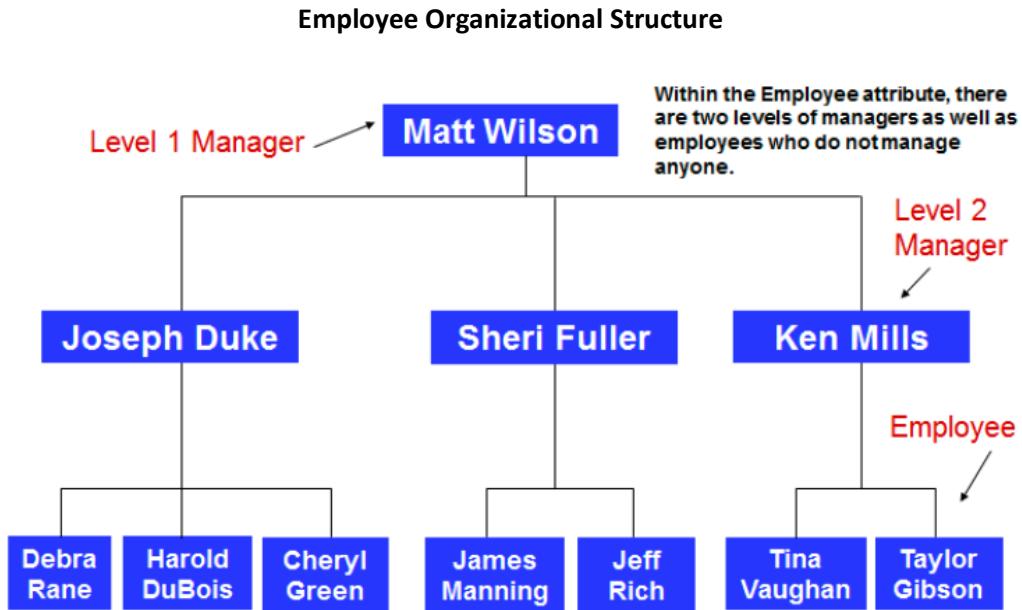
Recursive Hierarchies

A recursive hierarchy is one in which elements of an attribute have a parent-child relationship with other elements of the same attribute. All of the attributes in a hierarchy may be recursive, or a hierarchy may have only a single attribute that is recursive. For example, a company's organizational structure looks like the following:

Logical Data Model for Geography Hierarchy



At first, this hierarchy seems very simple and straightforward. However, within the Employee attribute, there are two levels of management along with the lowest-level employees who do not manage anyone. An organization chart for the company looks like the following:



In the database, the employee data is stored in a single table in a recursive fashion. This table looks like the following:

Lookup Table for Employee

LU_EMPLOYEE

The **Manager_ID** column references the employee IDs.

Employee_ID	Employee_Name	Manager_ID	Site_ID
1	Matt Wilson		1
2	Joseph Duke	1 (Matt Wilson)	1
3	Sheri Fuller	1 (Matt Wilson)	1
4	Ken Mills	1 (Matt Wilson)	1
5	Debra Rane	2 (Joseph Duke)	1
6	Harold DuBois	2 (Joseph Duke)	1
7	Cheryl Green	2 (Joseph Duke)	1
8	James Manning	3 (Sheri Fuller)	1
9	Jeff Rich	3 (Sheri Fuller)	1
10	Tina Vaughan	4 (Ken Mills)	1
11	Taylor Gibson	4 (Ken Mills)	1

The LU_EMPLOYEE table stores not only the ID and name of each employee, but it also has a Manager_ID column, which references the employee ID of each employee's manager. If you just want to view a list of all

employees, you can create an Employee attribute and map its ID and DESC forms to the Employee_ID and Employee_Name columns in the LU_EMPLOYEE table.

Generally, when you have a recursive attribute like Employee, you want to be able to run reports that show managers and their corresponding employees. In the database, the data for all three levels of employees come from the same columns in the same table. However, on a report, they are logically different attributes. For example, you could run a report that looks like the following:

Report with Manager and Employee Attributes

Report View: 'Local Template'			Switch to:
Level 1 Manager	Level 2 Manager	Employee	

The Level 1 Manager and Level 2 Manager attributes represent the two levels of management, and the Employee attribute represents the lowest-level employees who do not manage anyone. Because all three attributes map to the same columns in the same lookup table, you need to be able to alias the table three times in the SQL to retrieve the employee name for each of the three attributes on the template. By default, the SQL Engine aliases a table only once.

To resolve this issue with recursive hierarchies, you need to flatten the recursive attribute, creating separate lookup tables or views for each level of recursion.

You cannot use explicit table aliasing to support recursive hierarchies. Although you could create two logical table aliases for the LU_EMPLOYEE table and map each of the manager levels to one of the table aliases, this solution does not work because all of the employee records are contained in the lookup table that is aliased. When you run a report with any one of the three attributes (Level 1 Manager, Level 2 Manager, or Employee), it displays every employee in the table for each attribute.

Sometimes, tables with a recursive structure also contain a level column, which indicates the level of an element in the recursive hierarchy. For example, a Level 1 Manager would be "1," a Level 2 Manager would be "2," and an Employee would be "3." Using a level column like this inside a table does not resolve issues with recursive hierarchies because the MicroStrategy SQL Engine does not look at the specific data elements contained in the rows of a table.

Flattening a Recursive Hierarchy

To flatten the recursive LU_EMPLOYEE table, you need to create three separate lookup tables or views. The three tables or views look like the following:

Flattened Lookup Tables

LU_LEVEL1MANAGER

Lev1_Mgr_ID	Lev1_Mgr_Name	Site_ID
1	Matt Wilson	1

LU_LEVEL2MANAGER

Lev2_Mgr_ID	Lev2_Mgr_Name	Lev1_Mgr_ID
2	Joseph Duke	1 (Matt Wilson)
3	Sheri Fuller	1 (Matt Wilson)
4	Ken Mills	1 (Matt Wilson)

LU_EMPLOYEE

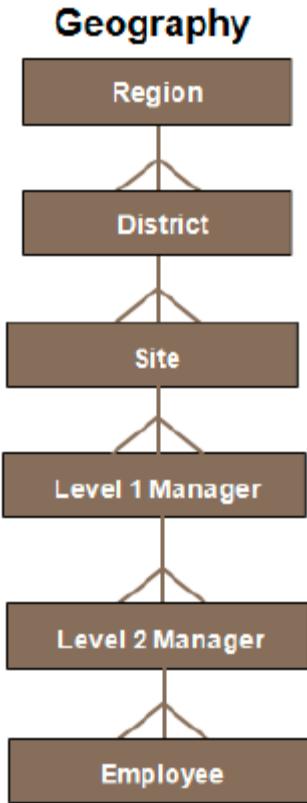
Employee_ID	Employee_Name	Lev2_Mgr_ID
5	Debra Rane	2 (Joseph Duke)
6	Harold DuBois	2 (Joseph Duke)
7	Cheryl Green	2 (Joseph Duke)
8	James Manning	3 (Sheri Fuller)
9	Jeff Rich	3 (Sheri Fuller)
10	Tina Vaughan	4 (Ken Mills)
11	Taylor Gibson	4 (Ken Mills)

After flattening the LU_EMPLOYEE table, you map the ID and DESC forms of the three attributes as follows:

- Level 1 Manager—Maps to the Lev1_Mgr_ID and Lev1_Mgr_Name columns in the LU_LEVEL1MANAGER table and the Lev1_Mgr_ID column in the LU_LEVEL2MANAGER table
- Level 2 Manager—Maps to the Lev2_Mgr_ID and Lev2_Mgr_Name columns in the LU_LEVEL2MANAGER table and the Lev2_Mgr_ID column in the LU_EMPLOYEE table
- Employee—Maps to the Employee_ID and Employee_Name columns in the LU_EMPLOYEE table

You can then change the data model to reflect the relationships between the three attributes as follows:

Revised Geography Logical Data Model



Now, if you run the report, the result set looks like the following:

Report Result with Recursive Relationships

Level 1 Manager	Level 2 Manager	Employee
Matt Wilson	Joseph Duke	Debra Rane Harold DuBois Cheryl Green
	Sheri Fuller	James Manning Jeff Rich
	Ken Mills	Tina Vaughan Taylor Gibson

Since the Level 1 Manager, Level 2 Manager, and Employee attributes map to different tables or views, they each are aliased in the SQL, which enables the query to display the employees with respect to the managerial relationships that exist. The SQL for this report looks like the following:

```
select a12.[Level1_Mgr_ID] AS Level1_Mgr_ID,  
a13.[Level1_Mgr_Name] AS Level1_Mgr_Name,
```

```

a11.[Level2_Mgr_ID] AS Level2_Mgr_ID,
a12.[Level2_Mgr_Name] AS Level2_Mgr_Name,
a11.[Employee_ID] AS Employee_ID,
a11.Employee_Name] AS Employee_Name
from [LU_EMPLOYEE] a11,
[LU_LEVEL2MANAGER] a12,
[LU_LEVEL1MANAGER] a13
where a11.[Level2_Mgr_ID] =
a12.[Level2_Mgr_ID] and
a12.[Level1_Mgr_ID] = a13.[Level1_Mgr_ID]

```

In the FROM clause, the SQL Engine uses the LU_LEVEL1MANAGER, LU_LEVEL2MANAGER, and LU_EMPLOYEE tables that comprise the flattened schema to retrieve the data for the result set.

In the above example, if you also have fact tables in your data warehouse that store data at the Employee level, then you could have resolved this issue using a completely denormalized lookup table as shown below:

Completely Denormalized Employee Lookup Table

LU_EMPLOYEE						
	Employee		Level2Manager		Level1Manager	
	Employee ID	Employee Name	Level2 Mgr ID	Level2 Mgr Name	Level1 Mgr ID	Level1 Mgr Name
5	Debra Rane	2	Joseph Duke	1	Matt Wilson	1
6	Harold DuBols	2	Joseph Duke	1	Matt Wilson	1
7	Cheryl Green	2	Joseph Duke	1	Matt Wilson	1
8	James Manning	3	Sheri Fuller	1	Matt Wilson	1
9	Jeff Rich	3	Sheri Fuller	1	Matt Wilson	1
10	Tina Vaughan	4	Ken Mills	1	Matt Wilson	1
11	Taylor Gibson	4	Ken Mills	1	Matt Wilson	1

Handling Complexities in Recursive Hierarchies

In the previous example, flattening the recursive table is a good solution since the number of levels is relatively small and fixed. Furthermore, there was an added assumption that the fact tables stored data at the employee level only.

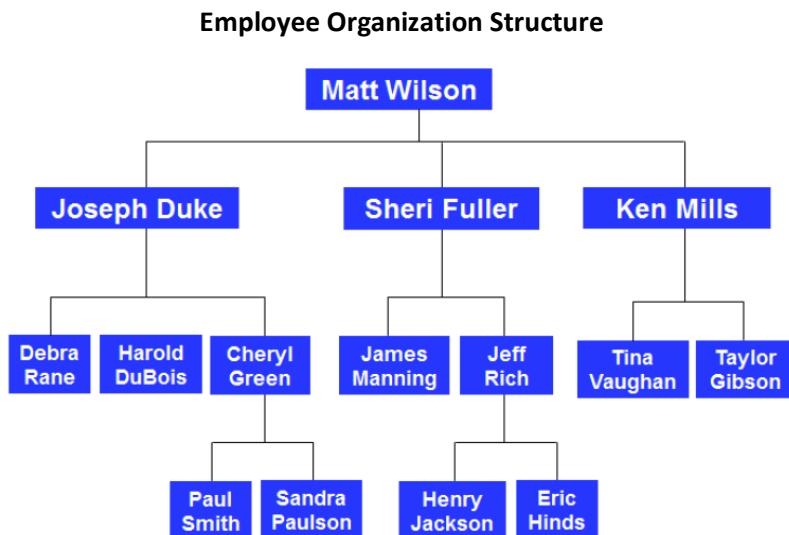
This then allows you to drill up and down the hierarchy. However, there could be situations where the fact tables contain data from higher levels. For instance, let us assume the above example represents a service organization. Both the level 1 and 2 managers along with the employees who report to them perform billable work. The billable hours are recorded in one fact table. So if Joseph Duke performed 20 hours of billable work for a customer, in order to create an accurate monthly billing report, he would have to be an employee in the employee table reporting to Matt Wilson who would be listed as a level 2 manager. Similarly, if Matt Wilson also performed billable work, then he would have to be a level 2 manager and an employee and in both cases reporting to himself.

Alternative Solution - Using a Relationship Table

Additional challenges posed by recursive hierarchies are that they could be ragged. Also, some recursive hierarchies could have no predefined limit to the number of levels.

There are different solutions possible based on whether there is a need to see separate attributes. In the example, it made sense to model three separate attributes so that you could look at business facts from an organizational hierarchy perspective. If such a requirement is not needed, then the entire hierarchy could be modelled through employee and manager attributes. Basically, you create a separate relationship table that links any employee to her direct and indirect managers. In addition, in the relationship table, there is another attribute that represents the distance that any employee is from the top of the hierarchy.

Essentially, the relationship table captures information in such a way that it effectively represents the parent-child relation in the hierarchy. Here is a sample organization structure showing the employees and their relationship to each other:



In our example, for Paul Smith the relationship table would contain multiple entries, capturing all of Paul's managers.

- Cheryl Green is Paul's direct manager (distance 1)
- Joseph Duke is Paul's indirect manager (distance 2)
- Matt Wilson is Paul's indirect manager (distance 3)

So for Paul, there would be three records in the relationship table, each indicating the overall relationship in the hierarchy. It may also be useful to add another row indicating a relationship from Paul to himself with a distance of zero. This will prove useful when the fact tables include data for both managers and employees. The following image shows the relationship table structure and the number of records for Paul:

Hierarchy Relationship Table

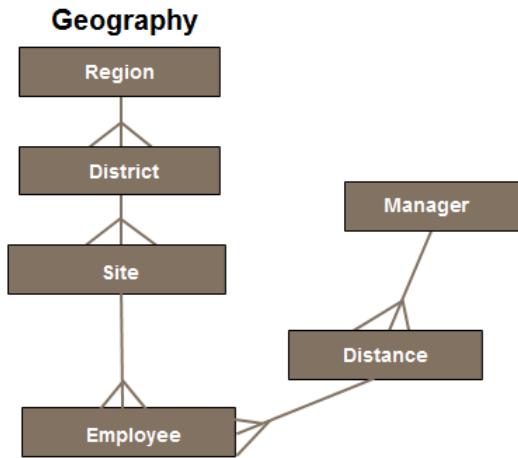
Employee_ID	Manager_ID	Distance	REL_HIERARCHY
...	
12 (Paul Smith)	12 (Paul Smith)	0	
12 (Paul Smith)	7 (Cheryl Green)	1	
12 (Paul Smith)	2 (Joseph Duke)	2	
12 (Paul Smith)	1 (Matt Wilson)	3	
...	

After creating the REL_HIERARCHY table above, you need to perform the following steps:

1. Add the REL_HIERARCHY table to the project.
2. Create the Employee attribute by mapping it to Employee_ID column in the LU_EMPLOYEE and REL_HIERARCHY tables.
3. Create the description form for the Employee attribute using Employee_Name column in the LU_EMPLOYEE table.
4. Use explicit table aliasing to create the LU_EMPLOYEE_ALIAS table.
5. Create the Manager attribute by mapping it to the Manager_ID in the REL_HIERARCHY table.
6. Use heterogeneous mapping to map the Employee_ID in the LU_EMPLOYEE_ALIAS table to the Manager attribute.
7. Create the description form for the Manager attribute using the Employee_Name column in the LU_EMPLOYEE_ALIAS table.
8. Create the Distance attribute by mapping it to the Distance_ID in the REL_HIERARCHY table.
9. Make Manager the parent and Employee the child of the Distance attribute using the REL_HIERARCHY table.
10. Update the project schema.

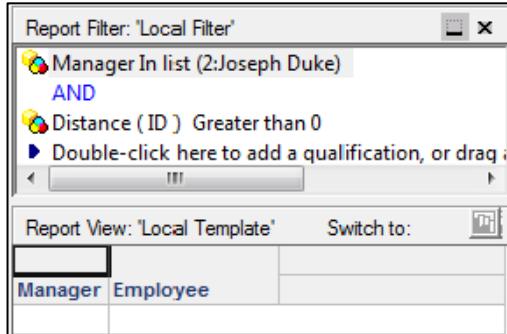
The following image shows the revised data model for the Geography hierarchy:

Revised Geography Logical Data Model



Now, if you wanted to see all the employees who report to Joseph Duke, you can create the following report:

Report for all Employees of a Specific Manager



When you run the report the following results are displayed:

Report Result of all Employees Reporting to a Specific Manager

Manager	Employee
Joseph Duke	Debra Rane
	Harold Dubois
	Cheryl Green
	Paul Smith
	Sandra Paulson

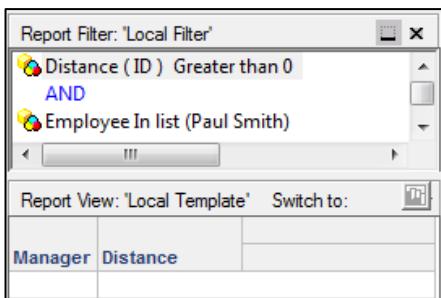
The SQL for the above report looks like the following:

```
select a12.[Manager_ID] AS Manager_ID,  
a13.[Employee_Name] AS Employee_Name,  
a11.[Employee_ID] AS Employee_ID,  
a11.[Employee_Name] AS Employee_Name0  
from [LU_EMPLOYEE] a11,  
[REL_HIERARCHY] a12,  
[LU_EMPLOYEE] a13  
where a11.[Employee_ID] = a12.[Employee_ID] and  
a12.[Manager_ID] = a13.[Employee_ID]  
and (a12.[Distance] > 0  
and a12.[Manager_ID] in (2))
```

Notice that the relationship table is used to retrieve all employees and managers who have a relationship with a distance greater than zero.

Sometimes you may not want to query a specific branch of the hierarchy, but rather for any given employee you want to find the chain of managers. The relationship table makes this possible and if you include the Distance attribute in the report template, you can then sort the result set to see an employee's entire reporting structure. For example, the following image shows the report template to determine all the managers for Paul Smith:

Report for all Managers for a Specific Employee



When you run the report the following results are displayed:

Report Result Showing Reporting Structure for a Specific Employee

Report details	
Report Filter: (Distance (ID) > 0) And (Employee = Paul Smith)	
Manager	Distance
Cheryl Green	1
Joseph Duke	2
Matt Wilson	3

The SQL for the above report looks like the following:

```
select distinct a12.[Manager_ID] AS Manager_ID,
a13.[Employee_Name] AS Employee_Name,
a12.[Distance] AS Distance
from [REL_HIERARCHY] a12,
[LU_EMPLOYEE] a13
where a12.[Manager_ID] = a13.[Employee_ID]
and (a12.[Employee_ID] in (12)
and a12.[Distance] > 0)
```

Notice that the relationship table is used to retrieve all the managers and their distances for Paul Smith.

Finally, what if you wanted to include in your report business fact data, such as the hours billed by each employee. The following image shows the fact table:

Fact Table Structure

FACT_EMPLOYEE_BILLING_HOURS		
Employee_ID	Billed_Hours	Bill_Date
...
2 (Joseph Duke)	5	11/5/2012
3 (Sheri Fuller)	7	11/6/2012
4 (Ken Mills)	5	11/9/2012
5 (Debra Rane)	8	11/5/2012
5 (Debra Rane)	7	11/6/2012
5 (Debra Rane)	6	11/7/2012
6 (Harold DuBois)	10	11/5/2012
6 (Harold DuBois)	10	11/6/2012
6 (Harold DuBois)	7	11/7/2012
7 (Cheryl Green)	11	11/5/2012
...

1. Add the FACT_EMPLOYEE_BILLING_HOURS table to the project.
2. Create the Billed Hours fact by mapping it to Billed_Hours column.
3. Edit the Employee attribute and make sure that the Employee_ID attribute form is mapped to the FACT_EMPLOYEE_BILLING_HOURS table.
4. Update the project schema.

5. Create Billed Hours metric.

Then, you could run a report that shows the total number of hours billed by all employees who are part of Joseph Duke's reporting chain. The result set looks like the following:

Result Set for Employee Information with a Metric

Employee	Metrics	Billed Hours
Joseph Duke		5
Debra Rane		21
Harold Dubois		27
Cheryl Green		27
Paul Smith		8
Sandra Paulson		6

Notice that the report displays correctly all the employees who are under Joseph Duke's chain of command. The report also shows the hours billed by Joseph himself. To exclude him from the report result, an additional report filter using the Distance attribute (Distance > 0) can be applied to this report.

The SQL for this report looks like the following:

```
select a11.[Employee_ID] AS Employee_ID,  
max(a13.[Employee_Name]) AS Employee_Name,  
sum(a11.[Billed_Hours]) AS WJXBFS1  
from FACT_EMPLOYEE_BILLING_HOURS] a11,  
[REL_HIERARCHY] a12,  
[LU_EMPLOYEE] a13  
where a11.[Employee_ID] = a12.[Employee_ID] and  
a11.[Employee_ID] = a13.[Employee_ID]  
and a12.[Manager_ID] in (2)  
group by a11.[Employee_ID]
```

Notice how the relationship table is used to join to the fact table to calculate the employees billed hours. The relationship table is also used to filter the report for only one manager and in this case Joseph Duke.

Exercises:

You should complete the following exercises using various projects found in the Advanced Data Warehousing project source. Instructions at the beginning of each section reference the projects that you need to use for that particular exercise.

Ragged Hierarchies Overview

For the exercises in this section, you will use the Ragged Hierarchies Project in the Advanced Data Warehousing project source. The hierarchies and schema for this project are included with the exercises. You need to review this information before beginning the exercises.

During the exercises, you will first create and run reports in the project without having any solution in place to resolve gaps in a ragged hierarchy. Then, you will see how changing the VLDB property at the attribute and report levels addresses the issue of aggregating data for skipped levels. Next you will see how revising the data model does not completely solve all of the issues caused by the ragged hierarchy. Finally, you will resolve the ragged hierarchy by populating the gaps in the hierarchy with system-generated values.

Ragged Hierarchies Project Information

The Ragged Hierarchies Project is based on the following logical data model:

Ragged Hierarchies Logical Data Model



In this data model, gaps exist in the data at the market level. Most account executives are assigned to a market, which then rolls up into a region.

However, two of the account executives have large enough client accounts that they form their own “markets.” Therefore, they roll up directly into a region as follows:

Gaps in Sales Organization Data



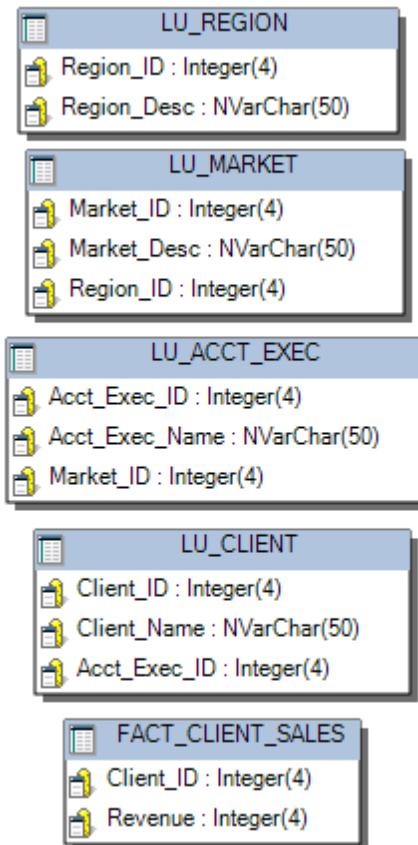
The Sales hierarchy looks like the following:

Sales Hierarchy



The schema for this project consists of the following five tables:

Ragged Hierarchies Schema

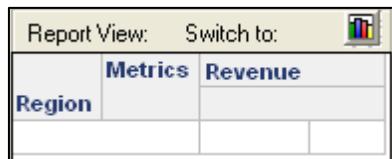


Ragged Hierarchies Exercises

Create and Run Reports with a Ragged Hierarchy

Create a report

1. In MicroStrategy Developer, in the Advanced Data Warehousing project source, open the Ragged Hierarchies Project.
2. In the Public Objects folder, in the Reports folder, create the following report:



You can access the Region attribute from the Sales hierarchy. The Revenue metric is located in the Metrics folder.

Run the report

3. Run the report. The result set should look like the following:

Region	Metrics	Revenue
East		\$662,093
Midwest		\$1,418,250
Mountain		\$732,953
Pacific		\$960,919

Drill on the report

4. Drill down on the entire report from Region to Market and choose to keep the parent attribute when drilling. The result set should look like the following:

You can use the Drill option on the Data menu to perform the drill while keeping the parent attribute.

Region	Market	Metrics	Revenue
East	Fashion - East		\$40,397
	Food - East		\$133,821
	Transportation - East		\$160,678
	Retail - East		\$327,197
Midwest	Fashion - Midwest		\$47,845
	Food - Midwest		\$485,207
	Transportation - Midwest		\$871,290
	Retail - Midwest		\$13,908
Mountain	Fashion - Mountain		\$76,390
	Food - Mountain		\$123,550
	Transportation - Mountain		\$478,901
	Retail - Mountain		\$54,112
Pacific	Fashion - Pacific		\$457,197
	Food - Pacific		\$234,590
	Transportation - Pacific		\$91,489
	Retail - Pacific		\$177,643

Notice that the markets related to each region (Food, Fashion, Transportation, and Retail) are included in the result set.

5. On the drill report, drill down on the entire report from Market to Account Executive, again choosing to keep the parent attribute when drilling. The result set should look like the following:

Region	Market	Account Executive	Metrics	Revenue
East	Fashion - East	Ed Diller	\$17,654	
		Karen Fay	\$22,743	
	Food - East	Travis Dent	\$87,921	
		Jen Fisher	\$45,900	
	Transportation - East	Al Cunta	\$126,008	
		Nelson Lee	\$34,670	
Midwest	Retail - East	Allen Green	\$327,197	
	Fashion - Midwest	Reggie Benson	\$47,845	
	Food - Midwest	Carl Smith	\$468,198	
		Dolores Goodwin	\$17,009	
	Transportation - Midwest	Tim Wesson	\$871,290	
Mountain	Retail - Midwest	Tom Watson	\$13,908	
	Fashion - Mountain	Terese Winn	\$76,390	
	Food - Mountain	Sam Hill	\$90,761	
		Dan Foster	\$32,789	
	Transportation - Mountain	Nate Brown	\$478,901	
Pacific	Retail - Mountain	Amanda Wood	\$54,112	
	Fashion - Pacific	Terry Olsen	\$457,197	
	Food - Pacific	Tina Sanford	\$234,590	
	Transportation - Pacific	Jim Wells	\$91,489	
	Retail - Pacific	Marlene Detrick	\$108,741	
		Chris Cooper	\$68,902	

Notice that Sara Kaplan (the account executive for the East region who is not assigned to a market) and Wes Vinson (the account executive for the Mountain region who is not assigned to a market) do not display on this report.

Why do they not display in the result set? The answer is in the SQL.

- Switch to the SQL View for this report. You should see the following SQL:

```

select    a14.[Region_ID] AS Region_ID,
          max(a15.[Region_Desc]) AS Region_Desc,
          a13.[Market_ID] AS Market_ID,
          max(a14.[Market_Desc]) AS Market_Desc,
          a12.[Acct_Exec_ID] AS Acct_Exec_ID,
          max(a13.[Acct_Exec_Name]) AS Acct_Exec_Name,
          sum(a11.[Revenue]) AS WJXBFS1
from      [FACT_CLIENT_SALES]      a11,
          [LU_CLIENT]           a12,
          [LU_ACCT_EXEC]        a13,
          [LU_MARKET]           a14,
          [LU_REGION]           a15
where     a11.[Client_ID] = a12.[Client_ID] and
          a12.[Acct_Exec_ID] = a13.[Acct_Exec_ID] and
          a13.[Market_ID] = a14.[Market_ID] and
          a14.[Region_ID] = a15.[Region_ID]
group by  a14.[Region_ID],
          a13.[Market_ID],
          a12.[Acct_Exec_ID]
  
```

To access account executive information, the join goes through the LU_MARKET table to relate Region to Account Executive. Since these two account executives are not assigned to markets, no relationship exists.

Therefore, the result set excludes them.

7. Close both drill reports, but leave the original report you created open. You do not need to save the drill reports.

Place a subtotal on the report

8. On the original report, place a Total subtotal. The total should display as follows:

Region	Metrics	Revenue
East		\$662,093
Midwest		\$1,418,250
Mountain		\$732,953
Pacific		\$960,919
Total		\$3,774,215

Notice that according to this report, the total revenue across all regions is \$3,774,215.

Save the report

9. Save this report in the Reports folder as Revenue by Region and close the report.

Create a second report

10. In the Reports folder, create the following report:

Report	Switch to:	
Client	Metrics	Revenue

You can access the Client attribute from the Sales hierarchy.

Run the report

11. Run the report. The result set should look like the following:

Client	Metrics	Revenue
Calvin Klein		\$17,654
Tommy Hilfiger		\$22,743
Kraft		\$87,921
Campbells		\$45,900
General Motors		\$126,008
Amtrak		\$34,670
Walmart		\$237,821
CVS		\$89,376
American Express		\$5,000,789
Levi-Strauss		\$47,845
Food Lion		\$468,198
Giant Food		\$17,009
United Airlines		\$871,290
Target		\$13,908
Aigner		\$76,390
Safeway		\$90,761
United Food Corporation		\$32,789
Southwest Airlines		\$478,901
Dillards		\$54,112
Citibank		\$3,045,752
VISA		\$7,931,870
Vera Wang		\$457,197
Groceries Galore		\$234,590
Northwest Cruises		\$91,489
True North		\$108,741
Patagonia		\$68,902

Notice that American Express, Citibank, and VISA (the clients that Sara Kaplan and Wes Vinson represent) are all included in the result set.

Place a subtotal on the report

12. Place a Total subtotal on the report. The total should display as follows:

Client	Metrics	Revenue
Calvin Klein		\$17,654
Tommy Hilfiger		\$22,743
Kraft		\$87,921
Campbells		\$45,900
General Motors		\$126,008
Amtrak		\$34,670
Walmart		\$237,821
CVS		\$89,376
American Express		\$5,000,789
Levi-Strauss		\$47,845
Food Lion		\$468,198
Giant Food		\$17,009
United Airlines		\$871,290
Target		\$13,908
Aigner		\$76,390
Safeway		\$90,761
United Food Corporation		\$32,789
Southwest Airlines		\$478,901
Dillards		\$54,112
Citibank		\$3,045,752
VISA		\$7,931,870
Vera Wang		\$457,197
Groceries Galore		\$234,590
Northwest Cruises		\$91,489
True North		\$108,741
Patagonia		\$68,902
Total		\$19,752,626

Notice that the true total for all of the revenue is \$19,752,626. That is a very different figure than you saw on the Revenue by Region report. The FACT_CLIENT_SALES table is keyed based on the client ID. Therefore, on this report, all clients are accounted for in the revenue figures. However, the Revenue by Region report joins to the FACT_CLIENT_SALES table through the LU_MARKET table.

Because of the gaps that exist at the market level, not all clients are accounted for in the aggregation.

Save the report

13. Save this report in the Reports folder as Revenue by Client and close the report.

Resolve the Ragged Hierarchy by Changing VLDB Properties

One method for resolving some of the issues caused by the gaps in this ragged hierarchy is to change the join VLDB Property for the Market attribute so that data can be aggregated from the skipped level.

Modify the Market and Region attributes

1. In the Schema Objects folder, in the Attributes folder, open the Market attribute.
2. On the Tools menu, select VLDB Properties.
3. Expand the Joins folder and select Preserve all final pass result elements.
4. Clear the Use default inherited value - (Default Settings) check box, if not already clear.
5. Click Preserve all elements of final pass result table with respect to lookup table but not relationship table, if not already selected.
6. Click Save and Close twice.
7. Update the schema.

Create Revenue by Market report

8. In the Public Objects folder, in the Reports folder, create the following report:

Report View: 'Local' Switch to:		
	Metrics	Revenue
Market		

You can access the Market attribute from the Sales hierarchy. The Revenue metric is located in the Metrics folder.

9. On the Data menu, select VLDB Properties.
10. Expand the Joins folder and select Preserve all final pass result elements.
11. Clear the Use default inherited value - (Default Settings) check box.
12. Click Do not listen to per report level setting, preserve elements of the final pass according to the setting at the attribute level. If this choice is selected at attribute level, it will be treated as preserve common elements (i.e. choice 1).
13. In the Joins folder, select Join Type.
14. Clear the Use default inherited value - (Default Settings) check box.
15. Click Join 92.
16. Save the report as Revenue by Market and close the report.

Run the report

17. Run the Revenue by Market report. Place a Total subtotal on the report.

The result set should look like the following:

Market		
		\$15,978,411
Fashion - East		\$40,397
Food - East		\$133,821
Transportation - East		\$160,678
Retail - East		\$327,197
Fashion - Midwest		\$47,845
Food - Midwest		\$485,207
Transportation - Midwest		\$871,290
Retail - Midwest		\$13,908
Fashion - Mountain		\$76,390
Food - Mountain		\$123,550
Transportation - Mountain		\$478,901
Retail - Mountain		\$54,112
Fashion - Pacific		\$457,197
Food - Pacific		\$234,590
Transportation - Pacific		\$91,489
Retail - Pacific		\$177,643
Total		\$19,752,626

Notice that the report shows the correct revenue total of \$19,752,626.

Why is the total correct on this report? The answer is in the SQL.

18. Switch to the SQL View for this report. You should see the following SQL:

Because the lookup tables can now left join with the fact table the report shows the correct revenue total. At first glance, this solution seems to have resolved the issues with ragged hierarchies. However, it has really only resolved the issue for a very specific reporting scenario—any time you want to aggregate the data at the Market level. If you run any query that involve agents of the Market attribute, the aggregation will not yield the correct total. To see the issue that is still present, perform the following steps:

Drill on the report

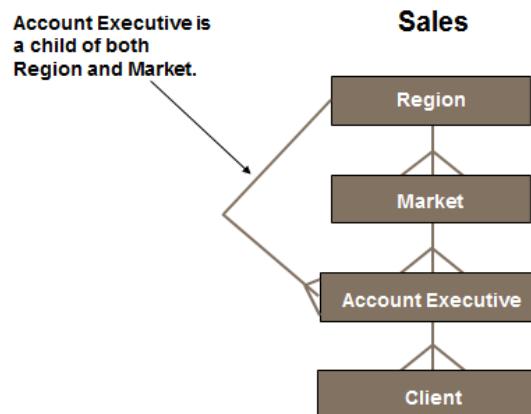
19. Switch to Grid View.
 20. Drill up on the entire report from Market to Region and choose to keep the parent attribute when drilling. Move Region to the left of Market. The result set should look like the following:

Region	Market	Metrics	Revenue
East	Fashion - East	\$40,397	
	Food - East	\$133,821	
	Transportation - East	\$160,678	
	Retail - East	\$327,197	
	Total	\$662,093	
Midwest	Fashion - Midwest	\$47,845	
	Food - Midwest	\$485,207	
	Transportation - Midwest	\$871,290	
	Retail - Midwest	\$13,908	
	Total	\$1,418,250	
Mountain	Fashion - Mountain	\$76,390	
	Food - Mountain	\$123,550	
	Transportation - Mountain	\$478,901	
	Retail - Mountain	\$54,112	
	Total	\$732,953	
Pacific	Fashion - Pacific	\$457,197	
	Food - Pacific	\$234,590	
	Transportation - Pacific	\$91,489	
	Retail - Pacific	\$177,643	
	Total	\$960,919	
Total		\$3,774,215	

The report does not include Sara Kaplan and Wes Vinson revenue because this query uses the relationship between Region and Market to retrieve the result set. Since Sara and Wes do not relate to any market, there is no way to link them to the region that they belong to in the LU_REGION table.

Because of this, their data is excluded in this report. Now that you have seen the problem that is still present when you change the VLDB Properties, let us see how revising the data model addresses this issue.

Another method for resolving some of the issues caused by the gaps in this ragged hierarchy is to revise the data model so that Region is directly related to both Market and Account Executive as follows:



Because revising the data model entails changing the logical relationship of Region and Account Executive such that Account Executive is a child of Region, you must also relate the two attributes in the database itself. To do so, you must include the ID for Region in the LU_ACCT_EXEC table.

Modify the LU_ACCT_EXEC table

1. Open Microsoft SQL Server Management Studio from your machine.
2. Open the Adv. Data warehouse_WH database and execute the LU_ACCT_EXEC table. The original table data looks like the following:

	Acct_Exec_ID	Acct_Exec_Name	Market_ID
1	1	Ed Diller	1
2	2	Karen Fay	1
3	3	Travis Dent	2
4	4	Jen Fisher	2
5	5	Al Cunta	3
6	6	Nelson Lee	3
7	7	Allen Green	4
8	8	Sara Kaplan	NULL
9	9	Reggie Benson	6
10	10	Carl Smith	7
11	11	Dolores Goodwin	7
12	12	Tim Wesson	8
13	13	Tom Watson	9
14	14	Terese Winn	10
15	15	Sam Hill	11
16	16	Dan Foster	11
17	17	Nate Brown	12
18	18	Amanda Wood	13
19	19	Wes Vinson	NULL
20	20	Terry Olsen	15
21	21	Tina Sanford	16

3. To add the Region_ID column, write the below query or add the new column from the design view:
`alter table [Adv. Data Warehouse_WH].[dbo].[LU_ACCT_EXEC] add Region_ID Int`
4. To enter values in the Region_ID column, right click on the table name and select **Edit Top 200 Rows** option.
5. Type the following values into the Region_ID column:
You need to click the first cell of the Region_ID column to place the cursor inside the column before you can type the values.

Acct_Exec_ID	Acct_Exec_Na...	Market_ID	Region_ID
1	Ed Diller	1	1
2	Karen Fay	1	1
3	Travis Dent	2	1
4	Jen Fisher	2	1
5	Al Cunta	3	1
6	Nelson Lee	3	1
7	Allen Green	4	1
8	Sara Kaplan	NULL	1
9	Reggie Benson	6	2
10	Carl Smith	7	2
11	Dolores Goodwin	7	2
12	Tim Wesson	8	2
13	Tom Watson	9	2
14	Terese Winn	10	3
15	Sam Hill	11	3
16	Dan Foster	11	3
17	Nate Brown	12	3
18	Amanda Wood	13	3
19	Wes Vinson	NULL	3
20	Terry Olsen	15	4
21	Tina Sanford	16	4
22	Jim Wells	17	4
23	Marlene Detrick	18	4
24	Chris Cooper	18	4
NULL	NULL	NULL	NULL

6. Save the changes
7. Close the table.
8. You now need to make MicroStrategy Architect recognize the changes you made to the LU_ACCT_EXEC table and define the logical relationship between the Region and Account Executive attributes.

Update the LU_ACCT_EXEC table structure

9. In MicroStrategy Developer, in the Ragged Hierarchies Project, open the Warehouse Catalog.
10. In the Warehouse Catalog, update the LU_ACCT_EXEC table structure.
You access the option to update table structure by right-clicking the table name in the Warehouse Catalog.
11. Save and close the Warehouse Catalog.

Define the relationship between Region and Account Executive

12. In the Schema Objects folder, in the Attributes folder, open the Region attribute.
13. Add Account Executive as a child attribute.
14. Save and close the Region attribute.

Update the project schema

15. Update the project schema.

Run the Revenue by Region report

16. Run the Revenue by Region report again. The result set should look like the following:

Region	Metrics	Revenue
East		\$5,662,882
Midwest		\$1,418,250
Mountain		\$11,710,575
Pacific		\$960,919
Total		\$19,752,626

Notice that the report now shows the correct revenue total of \$19,752,626.

Why is the total now correct on this report? The answer is in the SQL.

17. Switch to the SQL View for this report. You should see the following SQL:

```

select    a13.[Region_ID] AS Region_ID,
          max(a14.[Region_Desc]) AS Region_Desc,
          sum(a11.[Revenue]) AS WJXBFS1
from      [FACT_CLIENT_SALES]      a11,
          [LU_CLIENT]           a12,
          [LU_ACCT_EXEC]        a13,
          [LU_REGION]           a14
where     a11.[Client_ID] = a12.[Client_ID] and
          a12.[Acct_Exec_ID] = a13.[Acct_Exec_ID] and
          a13.[Region_ID] = a14.[Region_ID]
group by  a13.[Region_ID]
  
```

Because Region is directly related to the Account Executive attribute, the FACT_CLIENT_SALES table can join to the LU_REGION table using the LU_ACCT_EXEC table. This join yields the correct revenue total.

Drill on the report

18. Switch to Grid View.
19. Drill down on the entire report from Region to Account Executive and choose to keep the parent attribute when drilling. The result set should look like the following:

Region	Account Executive	Metrics	Revenue
East	Ed Diller		\$17,654
	Karen Fay		\$22,743
	Travis Dent		\$87,921
	Jen Fisher		\$45,900
	Al Cunta		\$126,008
	Nelson Lee		\$34,670
	Allen Green		\$327,197
	Sara Kaplan		\$5,000,789
	Total		\$5,662,882
Midwest	Reggie Benson		\$47,845
	Carl Smith		\$468,198
	Dolores Goodwin		\$17,009
	Tim Wesson		\$871,290
	Tom Watson		\$13,908
	Total		\$1,418,250
Mountain	Terese Winn		\$76,390
	Sam Hill		\$90,761
	Dan Foster		\$32,789
	Nate Brown		\$478,901
	Amanda Wood		\$54,112
	Wes Vinson		\$10,977,622
Pacific	Total		\$11,710,575
	Terry Olsen		\$457,197
	Tina Sanford		\$234,590
	Jim Wells		\$91,489
	Marlene Detrick		\$108,741
	Chris Cooper		\$68,902
Total	Total		\$960,919
			\$19,752,626

Notice that both Sara Kaplan and Wes Vinson are part of this result set. Because Region and Account Executive are directly related, their respective lookup tables can be joined without having to go through the LU_MARKET table.

At first glance, this solution seems to have resolved the issues with ragged hierarchies. However, it has really only resolved the issues for a very specific join path—any time you join directly from the LU_REGION table to the

LU_ACCT_EXEC table. If you run any query that involves the LU_MARKET table, this “solution” does not remove the problems associated with joining through the LU_MARKET table. To see some of the issues that are still present, perform the following steps:

20. On the drill report, select the row for Sara Kaplan and drill up to Market.

This drill action should return the following result:

Report: Revenue by Region->Account Executive->Market
 Status: Execution complete
 The report returns no data. Please check the report and/or the view filter definition.
 Starting Time: 11:29:19

No data is returned because this query uses the relationship between Account Executive and Market to retrieve the result set. Since Sara Kaplan does not relate to any market, there is no data to display on the report. You can see how this join occurs by looking at the SQL.

- Switch to the SQL View for this report. You should see the following SQL:

```

Pass1 - Query Execution: 0:00:00.00
        Data Fetching and Processing: 0:00:00.00
        Data Transfer from Datasource(s): 0:00:00.00
        Other Processing: 0:00:00.02
insert into ZZTXELMX0BIRF000
select distinct r11.[Market_ID] AS Market_ID
from [LU_ACCT_EXEC] r11
where r11.[Acct_Exec_ID] in (8)

Pass2 - Query Execution: 0:00:00.00
        Data Fetching and Processing: 0:00:00.00
        Data Transfer from Datasource(s): 0:00:00.00
        Other Processing: 0:00:00.02
        Rows selected: 0
select pa14.[Market_ID] AS Market_ID,
       max(a15.[Market_Desc]) AS Market_Desc,
       sum(a11.[Revenue]) AS WJXBFS1
from [FACT_CLIENT_SALES] a11,
     [LU_CLIENT] a12,
     [LU_ACCT_EXEC] a13,
     [ZZTXELMX0BIRF000] pa14,
     [LU_MARKET] a15
where a11.[Client_ID] = a12.[Client_ID] and
      a12.[Acct_Exec_ID] = a13.[Acct_Exec_ID] and
      a13.[Market_ID] = pa14.[Market_ID] and
      pa14.[Market_ID] = a15.[Market_ID]
group by pa14.[Market_ID]

```

Notice that there are two passes of SQL. The first pass selects the markets to include in the result set from the LU_ACCT_EXEC table.

The WHERE clause has a condition that filters on Sara Kaplan's ID since you drilled only on Sara Kaplan in the report. The result of the first pass is inserted into a temporary table. The second pass then uses this temporary table in the SELECT and FROM clauses to retrieve the markets that display in the final result set. Since the ID for Sara Kaplan does not relate to any market ID, the first pass of SQL does not return any market IDs. Therefore, the second pass of SQL cannot return any data for the report.

- Close both drill reports. You do not need to save them.

Add the Market attribute to the report

- On the Revenue by Region report, switch to Design View.
- Add the Market attribute to the report as follows:

Report View: 'Local'			Switch to:
		Metrics	Revenue
Region	Market		

You can access the Market attribute from the Sales hierarchy.

Run the report

25. Run the report again. The result set should look like the following:

Region	Market	Metrics	Revenue
East	Fashion - East		\$40,397
	Food - East		\$133,821
	Transportation - East		\$160,678
	Retail - East		\$327,197
	Total		\$662,093
Midwest	Fashion - Midwest		\$47,845
	Food - Midwest		\$485,207
	Transportation - Midwest		\$871,290
	Retail - Midwest		\$13,908
	Total		\$1,418,250
Mountain	Fashion - Mountain		\$76,390
	Food - Mountain		\$123,550
	Transportation - Mountain		\$478,901
	Retail - Mountain		\$54,112
	Total		\$732,953
Pacific	Fashion - Pacific		\$457,197
	Food - Pacific		\$234,590
	Transportation - Pacific		\$91,489
	Retail - Pacific		\$177,643
	Total		\$960,919
Total			\$3,774,215

With the Market attribute on the report, notice that the report now shows the incorrect revenue total of \$3,774,215.

Why is the total once again incorrect on this report? The answer is in the SQL.

26. Switch to the SQL View for this report. You should see the following SQL:

```
select a13.[Region_ID] AS Region_ID,
       max(a15.[Region_Desc]) AS Region_Desc,
       a13.[Market_ID] AS Market_ID,
       max(a14.[Market_Desc]) AS Market_Desc,
       sum(a11.[Revenue]) AS WJXBFS1
  from [FACT_CLIENT_SALES]      a11,
       [LU_CLIENT]          a12,
       [LU_ACCT_EXEC]        a13,
       [LU_MARKET]          a14,
       [LU_REGION]          a15
 where a11.[Client_ID] = a12.[Client_ID] and
       a12.[Acct_Exec_ID] = a13.[Acct_Exec_ID] and
       a13.[Market_ID] = a14.[Market_ID] and
       a13.[Region_ID] = a15.[Region_ID]
group by a13.[Region_ID],
         a13.[Market_ID]
```

Notice that the FROM clause now includes the LU_MARKET table. Since the join to the fact table occurs through this table, the sales for Clients with account executives who do not roll up into a market are left out of the aggregation.

Drill on the report

27. Switch to Grid View.
28. Drill down on the entire report from Market to Account Executive and choose to keep the parent attribute when drilling. The first part of the result set should look like the following:

Region	Market	Account Executive	Metrics	Revenue
East	Fashion - East	Ed Diller		\$17,654
		Karen Fay		\$22,743
		Total		\$40,397
	Food - East	Travis Dent		\$87,921
		Jen Fisher		\$45,900
		Total		\$133,821
	Transportation - East	Al Curta		\$126,008
		Nelson Lee		\$34,870
		Total		\$160,678
	Retail - East	Allen Green		\$327,197
		Total		\$327,197
		Total		\$662,093
Midwest	Fashion - Midwest	Reggie Benson		\$47,845
		Total		\$47,845
		Carl Smith		\$468,198
	Food - Midwest	Dolores Goodwin		\$17,009
		Total		\$485,207
		Tim Wesson		\$871,290
		Total		\$871,290
	Retail - Midwest	Tom Watson		\$13,908
		Total		\$13,908
		Total		\$1,418,250
Mountain	Fashion - Mountain	Terese Winn		\$76,390
		Total		\$76,390
		Sam Hill		\$90,761
	Food - Mountain	Dan Foster		\$32,789
		Total		\$123,550
		Nate Brown		\$478,901
		Total		\$478,901
	Retail - Mountain	Amanda Wood		\$54,112
		Total		\$54,112
		Total		\$732,953
	Fashion - Pacific	Terry Olsen		\$457,197

Again, notice that Sara Kaplan does not display in the result set for the East region, and Wes Vinson does not display in the result set for the Mountain region. Because the drill joins through the Market attribute, and therefore the LU_MARKET table, these account executives are left out.

29. Close the drill report. You do not need to save it.
30. Close the Revenue by Region report. You do not need to save any changes to the report.

Now that you have seen the issues that are still present when you revise the data model, you are going to return the tables and attribute relationships to their original definitions and resolve the ragged hierarchy by populating the gaps with system-generated values.

Resolve the Ragged Hierarchy Using System-Generated Values

Remove Region_ID from the LU_ACCT_EXEC table

1. In MSSQL, in the Adv. Data Warehouse_WH table, right-click the LU_ACCT_EXEC table.
2. Select Design view

3. Right-click the Region_ID row header and select Delete Column field.

A screenshot of the Microsoft SQL Server Management Studio (SSMS) interface. A table named 'Region' is displayed with columns: Column Name, Data Type, and Allow Nulls. The 'Region_ID' column is selected, and a context menu is open over it. The menu options are: Set Primary Key, Insert Column, Delete Column (which is highlighted with a yellow background), Relationships..., Indexes/Keys..., Fulltext Index..., XML Indexes..., Check Constraints..., Spatial Indexes..., Generate Change Script..., and Properties. The 'Properties' option is at the bottom of the menu. The keyboard shortcut 'Alt+Enter' is shown to the right of the menu.

Column Name	Data Type	Allow Nulls
Acct_Exec_ID	int	<input checked="" type="checkbox"/>
Acct_Exec_Name	nvarchar(50)	<input checked="" type="checkbox"/>
Market_ID	int	<input checked="" type="checkbox"/>
Region_ID	int	<input checked="" type="checkbox"/>

4. Save the changes.
5. The table should now look like the following:

Results Messages

	Acct_Exec_ID	Acct_Exec_Name	Market_ID
1	1	Ed Diller	1
2	2	Karen Fay	1
3	3	Travis Dent	2
4	4	Jen Fisher	2
5	5	Al Cunta	3
6	6	Nelson Lee	3
7	7	Allen Green	4
8	8	Sara Kaplan	NULL
9	9	Reggie Benson	6
10	10	Carl Smith	7
11	11	Dolores Goodwin	7
12	12	Tim Wesson	8
13	13	Tom Watson	9
14	14	Terese Winn	10
15	15	Sam Hill	11
16	16	Dan Foster	11
17	17	Nate Brown	12
18	18	Amanda Wood	13
19	19	Wes Vinson	NULL
20	20	Terry Olsen	15

Enter system values in the LU_MARKET table

6. In MSSQL, in the Adv. Data Warehouse_WH table, right-click the LU_MARKET table and select **Select Top 1000 Rows** option. The original table data looks like the following:

	Market_ID	Market_Desc	Region_ID
1	1	Fashion - East	1
2	2	Food - East	1
3	3	Transportation - East	1
4	4	Retail - East	1
5	6	Fashion - Midwest	2
6	7	Food - Midwest	2
7	8	Transportation - Midwest	2
8	9	Retail - Midwest	2
9	10	Fashion - Mountain	3
10	11	Food - Mountain	3
11	12	Transportation - Mountain	3
12	13	Retail - Mountain	3
13	15	Fashion - Pacific	4
14	16	Food - Pacific	4
15	17	Transportation - Pacific	4
16	18	Retail - Pacific	4

7. Enter two new records at the bottom of the table as follows:

Market_ID	Market_Desc	Region_ID
1	Fashion - East	1
2	Food - East	1
3	Transportation ...	1
4	Retail - East	1
6	Fashion - Midwest	2
7	Food - Midwest	2
8	Transportation ...	2
9	Retail - Midwest	2
10	Fashion - Mountain	3
11	Food - Mountain	3
12	Transportation ...	3
13	Retail - Mountain	3
15	Fashion - Pacific	4
16	Food - Pacific	4
17	Transportation ...	4
18	Retail - Pacific	4
19	No Assigned Market	1
20	No Assigned Market	3
NULL	NULL	NULL

8. Save and close the table.

Enter market IDs in the LU_ACCT_EXEC table

9. In MSSQL, in the Adv. Data Warehouse_WH table, right-click the LU_ACCT_EXEC table and select **Edit Top 200 Rows** option.
10. For the row with Sara Kaplan, in the Market_ID column, type 19.
11. For the row with Wes Vinson, in the Market_ID column, type 20. When you have made these changes, the table should look like the following:

Acct_Exec_ID	Acct_Exec_Na...	Market_ID
1	Ed Diller	1
2	Karen Fay	1
3	Travis Dent	2
4	Jen Fisher	2
5	Al Cunta	3
6	Nelson Lee	3
7	Allen Green	4
8	Sara Kaplan	19
9	Reggie Benson	6
10	Carl Smith	7
11	Dolores Goodwin	7
12	Tim Wesson	8
13	Tom Watson	9
14	Terese Winn	10
15	Sam Hill	11
16	Dan Foster	11
17	Nate Brown	12
18	Amanda Wood	13
19	Wes Vinson	20
20	Terry Olsen	15
21	Tina Sanford	16
22	Jim Wells	17
23	Marlene Detrick	18
24	Chris Cooper	18
NULL	NULL	NULL

12. Save and close the table.

Return to the original attribute relationships

13. In MicroStrategy Developer, in the Ragged Hierarchies project, in the Schema Objects/Attributes folder, open the Region attribute.
14. Remove Account Executive as a child attribute.
15. Modify the ID form to remove LU_ACCT_EXEC as a source table.
16. Save and close the Region attribute.

Update the LU_ACCT_EXEC table structure

17. Open the Warehouse Catalog.

18. Update the LU_ACCT_EXEC table structure.
19. Save and close the Warehouse Catalog.

Update the project schema

20. Update the project schema.

Run the Revenue by Region report

21. Run the Revenue by Region report again. The result set should look like the following:

Region	Metrics	Revenue
East	\$5,862,882	
Midwest	\$1,418,250	
Mountain	\$11,710,575	
Pacific	\$960,919	
Total	\$19,752,626	

Notice that the total on the report is aggregating all of the revenue in the fact table.

22. Switch to the SQL View for the report. The SQL should look like the following:

```

select    a14.[Region_ID] AS Region_ID,
          max(a15.[Region_Desc]) AS Region_Desc,
          sum(a11.[Revenue]) AS WJXBFS1
from      [FACT_CLIENT_SALES]    a11,
          [LU_CLIENT]        a12,
          [LU_ACCT_EXEC]    a13,
          [LU_MARKET]       a14,
          [LU_REGION]       a15
where     a11.[Client_ID] = a12.[Client_ID] and
          a12.[Acct_Exec_ID] = a13.[Acct_Exec_ID] and
          a13.[Market_ID] = a14.[Market_ID] and
          a14.[Region_ID] = a15.[Region_ID]
group by  a14.[Region_ID]
  
```

Notice that the FROM clause contains the LU_MARKET table. This query joins through the LU_MARKET table, but because of the values you entered, all of the account executives are now related to a market and are included in the aggregation.

Drill on the report

23. Switch to Grid View.
24. Drill down on the entire report from Region to Market and choose to keep the parent attribute when drilling. The result set should look like the following:

Region	Market	Metrics	Revenue
East	Fashion - East		\$40,397
	Food - East		\$133,821
	Transportation - East		\$160,678
	Retail - East		\$327,197
	No Assigned Market - East		\$5,000,789
	Total		\$5,662,882
Midwest	Fashion - Midwest		\$47,845
	Food - Midwest		\$485,207
	Transportation - Midwest		\$871,290
	Retail - Midwest		\$13,908
	Total		\$1,418,250
	Total		\$11,710,575
Mountain	Fashion - Mountain		\$76,390
	Food - Mountain		\$123,550
	Transportation - Mountain		\$478,901
	Retail - Mountain		\$54,112
	No Assigned Market - Mountain		\$10,977,622
	Total		\$960,919
Pacific	Fashion - Pacific		\$457,197
	Food - Pacific		\$234,590
	Transportation - Pacific		\$91,489
	Retail - Pacific		\$177,643
	Total		\$19,752,626

Notice that the market values you entered display in the result set.

25. On the drill report, drill down on the entire report from Market to Account Executive and choose to keep the parent attribute when drilling. The first part of the result set should look like the following:

Region	Market	Account Executive	Metrics	Revenue
East	Fashion - East	Ed Diller	\$17,654	
		Karen Fay	\$22,743	
		Total	\$40,397	
	Food - East	Travis Dent	\$87,921	
		Jen Fisher	\$45,900	
		Total	\$133,821	
	Transportation - East	Al Cunta	\$126,008	
		Nelson Lee	\$34,670	
		Total	\$160,678	
	Retail - East	Allen Green	\$327,197	
		Total	\$327,197	
	No Assigned Market - East	Sara Kaplan	\$5,000,789	
		Total	\$5,000,789	
		Total	\$5,662,882	
Midwest	Fashion - Midwest	Reggie Benson	\$47,845	
		Total	\$47,845	
	Food - Midwest	Carl Smith	\$468,198	
		Dolores Goodwin	\$17,009	
		Total	\$485,207	
	Transportation - Midwest	Tim Wesson	\$871,290	
		Total	\$871,290	
		Tom Watson	\$13,908	
		Total	\$13,908	
		Total	\$1,418,250	
Mountain	Fashion - Mountain	Terese Winn	\$76,390	
		Total	\$76,390	
	Food - Mountain	Sam Hill	\$90,761	
		Dan Foster	\$32,789	
		Total	\$123,550	
	Transportation - Mountain	Nate Brown	\$478,901	
		Total	\$478,901	
		Amanda Wood	\$54,112	
		Total	\$54,112	
	No Assigned Market - Mountain	Wes Vinson	\$10,977,622	
		Total	\$10,977,622	

Notice that both Sara Kaplan and Wes Vinson are part of the result set.

26. Close all of the reports. You do not need to save the drill reports.

Split Hierarchies Overview

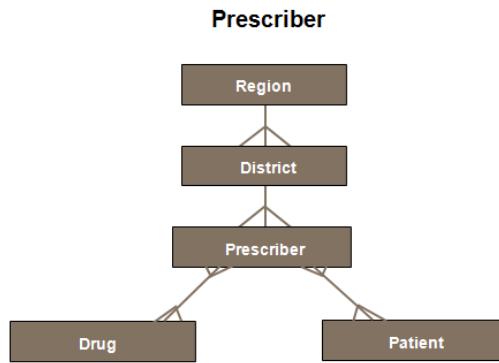
For the exercises in this section, you will use the Split Hierarchies Project in the Advanced Data Warehousing project source. The hierarchies and schema for this project are included with the exercises. You need to review this information before beginning the exercises.

During the exercises, you will first create and run reports in the project without having any solution in place so that you can see the effect of a split hierarchy on the result set. Then, you will resolve the split hierarchy by creating and implementing a joint child relationship.

Split Hierarchies Project Information

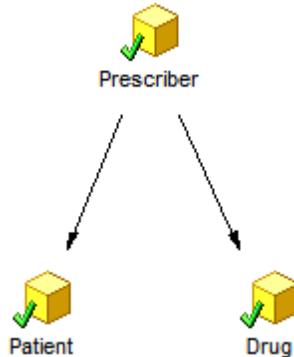
The Split Hierarchies Project is based on the following logical data model:

Split Hierarchies Logical Data Model



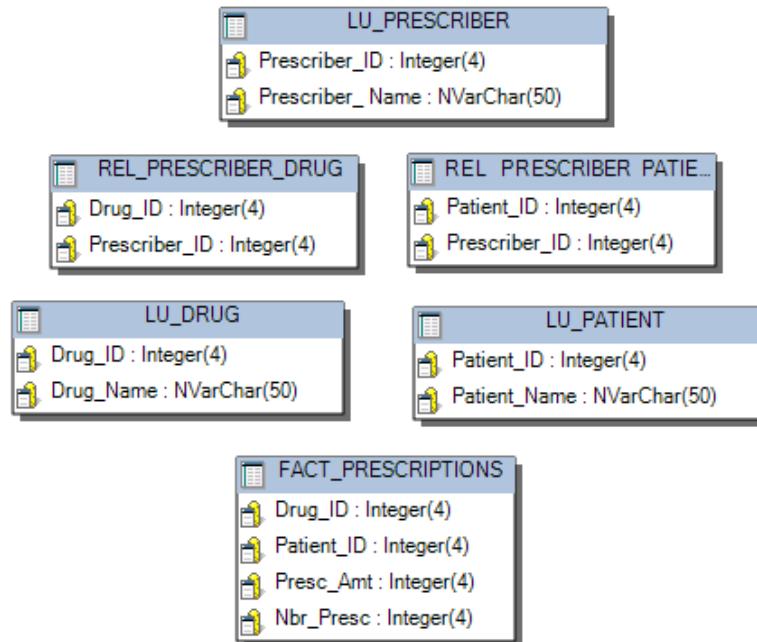
In the original data model, the hierarchy is split at its base by the Drug and Patient attributes. Because these exercises focus on these attributes and their parent attribute, the hierarchy exists in the project only up to the Prescriber attribute. The Prescriber hierarchy looks like the following:

Prescriber Hierarchy



The schema for this project consists of the following six tables:

Split Hierarchies Schema



Split Hierarchies Exercises

Create and Run Reports with a Split Hierarchy

Create a report with prescriber and drug information

1. In MicroStrategy Developer, in the MicroStrategy Analytics Modules project source, open the Split Hierarchies Project.
2. In the Public Objects folder, in the Reports folder, create the following report:

Report View: Switch to: <input type="button" value="TH"/>		
Prescriber	Drug	

You can access the Prescriber and Drug attributes from the Prescriber hierarchy.

Run the report

3. Run the report. The result set should look like the following:

Prescriber	Drug
	Zestril
Evan Thomas	Histussin
	Vasotec
Frank Hanford	Prozac
Tacia Hunt	Levaquin
	Histussin
Philip Sizemore	Prozac
Debra Peabody	Cipro
	Ceclor
Felicia Whitt	Cipro
	Prozac
Colby Sinclaire	Glynase
	Vasotec

This report displays each of the drugs that various prescribers have prescribed.

4. Switch to the SQL View for the report. The SQL should look like the following:

```

select    a12.[Prescriber_ID] AS Prescriber_ID,
          a13.[Prescriber_Name] AS [Prescriber_Name],
          a11.[Drug_ID] AS Drug_ID,
          a11.[Drug_Name] AS Drug_Name
from      [LU_DRUG]           a11,
          [REL_PRESCRIBER_DRUG] a12,
          [LU_PRESCRIBER]       a13
where     a11.[Drug_ID] = a12.[Drug_ID] and
          a12.[Prescriber_ID] = a13.[Prescriber_ID]
  
```

Notice in the FROM clause that the REL_PRESCRIBER_DRUG table joins the Prescriber and Drug attributes as you would expect since this table maps the relationship between them.

Save the report

5. Switch to Grid View.
6. Save this report in the Reports folder as Prescriber-Drug Report and close the report.

Create a report with prescriber and patient information

7. In the Reports folder, create the following report:

Report View:		Switch to:
Prescriber	Patient	

You can access the Patient attribute from the Prescriber hierarchy.

Run the report

8. Run the report. The result set should look like the following:

Prescriber	Patient
Evan Thomas	Cindy Tyler
	Henry Moore
	Frank Childress
Frank Hanford	Michael Helon
Tacia Hunt	Sam Tate
	Tina Drake
	Olivia Sanchez
	Jared Whitman
Philip Sizemore	Penelope Schnaitt
	Frank Childress
	Jared Whitman
Debra Peabody	Olivia Sanchez
	Bill Conley
	Angela Johnson
	Kim Arnold
Felicia Whitt	Penelope Schnaitt
	Michael Helon
Colby Sinclaire	Cody Abrams
	Rachel Swanson
	Ariana Fulson

This report displays each of the patients for whom the various prescribers have prescribed drugs.

- Switch to the SQL View for the report. The SQL should look like the following:

```

select    a12.[Prescriber_ID] AS Prescriber_ID,
          a13.[Prescriber_Name] AS [Prescriber_Name],
          a11.[Patient_ID] AS Patient_ID,
          a11.[Patient_Name] AS Patient_Name
from      [LU_PATIENT]      a11,
          [REL_PRESCRIBER_PATIENT]      a12,
          [LU_PRESCRIBER]a13
where     a11.[Patient_ID] = a12.[Patient_ID] and
          a12.[Prescriber_ID] = a13.[Prescriber_ID]
  
```

Notice in the FROM clause that the REL_PRESCRIBER_PATIENT table joins the Prescriber and Patient attributes as you would expect since this table maps the relationship between them.

Save the report

- Switch to Grid View.
- Save this report in the Reports folder as Prescriber-Patient Report and close the report.
- Create a report with prescriber and patient information and a metric
- In the Reports folder, create the following report:

Report View: 'Local Template'			Switch to:
Prescriber	Patient	Metrics	Prescription Amount

The Prescription Amount metric is located in the Metrics folder.

Run the report

13. Run the report. The result set should look like the following:

Prescriber	Patient	Metrics	Prescription Amount
Evan Thomas	Cindy Tyler		\$135
	Henry Moore		\$45
	Tina Drake		\$90
	Cody Abrams		\$35
	Frank Childress		\$150
Frank Hanford	Cody Abrams		\$50
	Michael Helon		\$200
	Ariana Fulson		\$150
	Jared Whitman		\$100
Tacia Hunt	Cindy Tyler		\$135
	Sam Tate		\$120
	Henry Moore		\$45
	Tina Drake		\$270
Philip Sizemore	Cody Abrams		\$50
	Michael Helon		\$200
	Ariana Fulson		\$150
	Jared Whitman		\$100
	Penelope Schnaidt		\$120
Debra Peabody	Olivia Sanchez		\$60
	Bill Conley		\$120
	Michael Helon		\$40
	Angela Johnson		\$220
	Kim Arnold		\$80
Felicia Whitt	Penelope Schnaidt		\$120
	Michael Helon		\$40
	Angela Johnson		\$40
	Kim Arnold		\$80
	Cody Abrams		\$85
Colby Sinclair	Rachel Swanson		\$60
	Michael Helon		\$200
	Frank Childress		\$70
	Ariana Fulson		\$150
	Jared Whitman		\$100

Compare the patients related to each prescriber in this result set with the patients related to each prescriber in the result set for the Prescriber-Patient Report.

Why do you see more patients associated with prescribers in this report than in the Prescriber-Patient Report? The answer is in the SQL.

14. Switch to the SQL View for the report. The SQL should look like the following:

```

select    a12.[Prescriber_ID] AS Prescriber_ID,
          max(a14.[Prescriber_Name]) AS [Prescriber_Name],
          a11.[Patient_ID] AS Patient_ID,
          max(a13.[Patient_Name]) AS Patient_Name,
          sum(a11.[Presc_Amt]) AS WJXBFS1
from      [FACT_PRESCRIPTIONS]    a11,
          [REL_PRESCRIBER_DRUG] a12,
          [LU_PATIENT]         a13,
          [LU_PRESCRIBER]a14
where     a11.[Drug_ID] = a12.[Drug_ID] and
          a11.[Patient_ID] = a13.[Patient_ID] and
          a12.[Prescriber_ID] = a14.[Prescriber_ID]
group by  a12.[Prescriber_ID],
          a11.[Patient_ID]

```

Notice in the FROM and WHERE clauses that the query joins the fact table to the LU_PATIENT table using the join path made available by the Drug attribute. As a result, this query uses the REL_PRESCRIBER_DRUG table to join to the LU_PRESCRIBER table, not the REL_PRESCRIBER_PATIENT table. Therefore, in the result set, you do not see patients related only to the prescribers who have prescribed medication to them. Instead, you see patients related to any prescriber who has ever prescribed a drug they have taken, regardless of whether they were ever a patient of that prescriber.

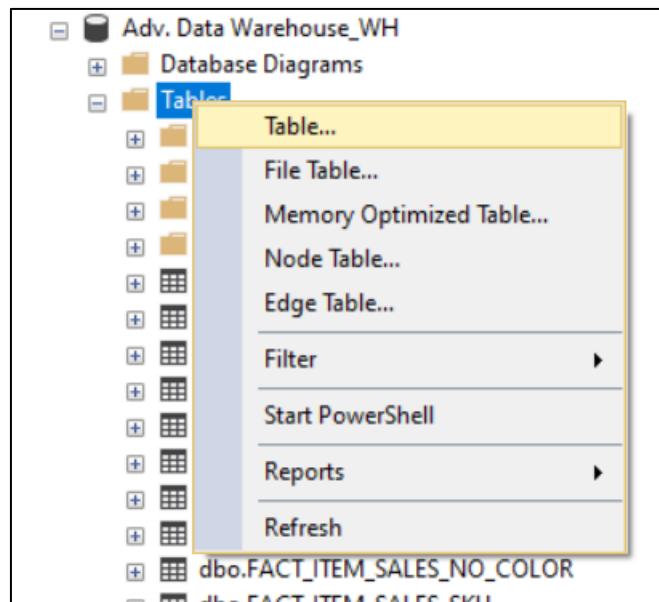
Save the report

15. Switch to Grid View.
16. Save this report in the Reports folder as Prescriber-Patient-Metric Report and close the report.

Resolve the Split Hierarchy Using a Joint Child Relationship

Create a relationship table

1. In the **Adv. Data Warehouse_WH** database, right click the folder name **Tables** and select **table**



2. This action displays an empty table.
In the table, create the following columns:

The image below also shows the appropriate data type to use for each column.

Column Name	Data Type	Allow Nulls
Prescriber_ID	int	<input checked="" type="checkbox"/>
Drug_ID	int	<input checked="" type="checkbox"/>
Patient_ID	int	<input checked="" type="checkbox"/>
		<input type="checkbox"/>

Click Save.

3. In the Save As window, in the Table Name box, type REL_PRESCRIBER_DRUG_PATIENT_DEMO as the table name.
4. Click OK.

At this point, you would enter the data; however, for this exercise this has already been done for you as shown in the table below:

```

***** Script for SelectTopNRows command from SSMS *****/
SELECT TOP (1000) [Prescriber_ID]
      ,[Drug_ID]
      ,[Patient_ID]
  FROM [Adv. Data Warehouse_WH].[dbo].[REL_PRESCRIBER_DRUG_PATIENT]

```

0 % < Results Messages

	Prescriber_ID	Drug_ID	Patient_ID
1	1	1	11
2	3	2	4
3	6	5	5
4	1	3	3
5	3	2	2
6	1	3	1
7	3	3	4
8	7	4	12
9	5	5	15
10	4	4	14
11	7	4	8
12	5	5	13
13	6	5	10
14	7	6	9
15	5	7	7
16	5	7	6
17	5	7	13
18	1	8	11
19	7	8	8
20	2	4	10

5. Close the REL_PRESCRIBER_DRUG_PATIENT_DEMO table.
6. In the Tables list, double-click the REL_PRESCRIBER_DRUG_PATIENT table.
Notice that that table structure and the data match the above image.
7. Close the table.

Add the relationship table to the project

8. In MicroStrategy Developer, in the Split Hierarchies project, open the Warehouse Catalog.
9. Add the REL_PRESCRIBER_DRUG_PATIENT table to the project.
10. Save and close the Warehouse Catalog.

Map attributes to relationship table

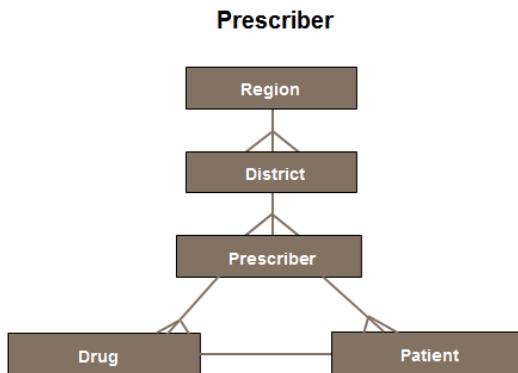
11. In the Schema Objects folder, in the Attributes folder, open the Prescriber attribute.
12. Remove Drug and Patient as children of the attribute.
13. Modify the ID form to remove REL_PRESCRIBER_DRUG and REL_PRESCRIBER_PATIENT as source tables.
14. Ensure that the REL_PRESCRIBER_DRUG_PATIENT table is selected as a source table.
15. Click OK.
16. Save and close Prescriber attribute.
17. In the Schema Objects folder, in the Attributes folder, open the Drug attribute.
18. Modify the ID form to remove REL_PRESCRIBER_DRUG as a source table.
19. Ensure that the REL_PRESCRIBER_DRUG_PATIENT table is selected as a source table.
20. Click OK.
21. Save and close Drug attribute.
22. In the Schema Objects folder, in the Attributes folder, open the Patient attribute.
23. Modify the ID form to remove REL_PRESCRIBER_PATIENT as a source table.
24. Ensure that the REL_PRESCRIBER_DRUG_PATIENT table is selected as a source table.
25. Click OK.
26. Save and close Patient attribute.

Create joint child relationship

27. Open the Prescriber attribute.
28. Add Drug and Patient as children of this attribute.
29. In the Add Children Attributes window, under Selected Children, select the Create as Joint Child check box.
30. Click OK.
31. Save and close Prescriber attribute.
32. Update the project schema.

In creating the joint child relationship, you are essentially changing the data model to look like the following:

Modified Logical Data Model for Prescriber Hierarchy



Run the Prescriber-Patient-Metric Report

33. Run the Prescriber-Patient-Metric Report again. The result set should look like the following:

Prescriber	Patient	Metrics	Prescription Amount
Evan Thomas	Cindy Tyler		\$135
	Henry Moore		\$45
	Frank Childress		\$150
Frank Hanford	Michael Helon		\$200
Tacia Hunt	Sam Tate		\$120
	Tina Drake		\$270
Philip Sizemore	Jared Whitman		\$100
	Olivia Sanchez		\$60
Debra Peabody	Bill Conley		\$120
	Angela Johnson		\$220
	Kim Arnold		\$80
Felicia Whitt	Penelope Schnaidt		\$120
	Michael Helon		\$40
Colby Sinclaire	Cody Abrams		\$85
	Rachel Swanson		\$60
	Ariana Fulson		\$150

The result set correctly relates the prescribers and their patients. You can compare this result set to the previous one for this same report earlier in the exercises.

34. Switch to the SQL View for the report. The SQL should look like the following:

```

select    a12.[Prescriber_ID] AS Prescriber_ID,
          max(a14.[Prescriber_Name]) AS [Prescriber_Name],
          a11.[Patient_ID] AS Patient_ID,
          max(a13.[Patient_Name]) AS Patient_Name,
          sum(a11.[Presc_Amt]) AS WJXBFS1
from      [FACT_PRESCRIPTIONS]      a11,
          [REL_PRESCRIBER_DRUG_PATIENT]      a12,
          [LU_PATIENT]      a13,
          [LU_PRESCRIBER]a14
where     a11.[Drug_ID] = a12.[Drug_ID] and
          a11.[Patient_ID] = a12.[Patient_ID] and
          a11.[Patient_ID] = a13.[Patient_ID] and
          a12.[Prescriber_ID] = a14.[Prescriber_ID]
group by  a12.[Prescriber_ID],
          a11.[Patient_ID]

```

Because of the joint child, the report now contains the correct result set since the information from the FACT_PRESCRIPTIONS table is joined to the prescriber information through the REL_PRESCRIBER_DRUG_PATIENT table.

35. Close the report.

Recursive Hierarchies Overview

This set of exercises contains two different recursive scenarios. You will find both them in the Recursive Hierarchies Project in the Advanced Data Warehousing project source. The hierarchies and schema for the two scenarios are included with the exercises. You need to review this information before beginning the exercises.

For the first scenario, you will first view table data that contains recursive relationships. Next, you will flatten the recursive table into three lookup tables.

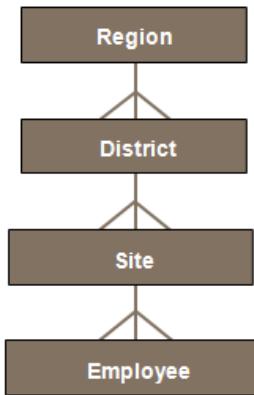
Finally, you will create and run a report using the flattened lookup tables.

Recursive Hierarchies Project Information

You will use the Recursive Hierarchies Project to build many of the project objects as part of the exercises. The image below shows the logical data model with the recursive attribute.

Recursive Hierarchies Logical Data Model

Geography



In this data model, the Employee attribute is the recursive attribute. Because these exercises focus on recursion, you will build attributes only for the employee data. You will modify the logical data model to separate the

Employee attribute into several logical attributes that reflect the recursive relationships.

Recursive Hierarchies Exercises

[View Recursive Data](#)

[View the recursive lookup table for employees](#)

1. In MSSQL, in the **Adv. Data Warehouse_WH** table, right-click the **LU_EMPLOYEE_RECUSION** table and select **Select Top 1000 Rows** option. The original table data looks like the following:

Employee_ID	Employee_Name	Manager_ID
1	Matt Wilson	NULL
2	Joe Thomburke	1
3	Mattie Nader	1
4	Tom Nelson	1
5	Sandra Cooper	2
6	Jason Kidd	2
7	Jeff Tazewell	2
8	Tanya Cummins	2
9	Mary Sikes	2
10	Michael Ebert	2
11	Cassie Henderson	2
12	Jill Thompson	3
13	Nate Landers	3
14	Alex Snow	3
15	Ted Mills	3
16	Andrea Cosby	3
17	Miriam Green	3
18	Rachel Porter	3
19	Guy Fenn	3
20	James DeVille	3
21	Serena Williams	4
22	Amy Murphy	4
23	David Robbins	4
24	Gloria Brenner	4
25	Andrew Radson	4
26	Anna Radev	4
27	Marie White	4
28	Ned Gordon	4
29	William Ashe	4
30	Katrina Ewell	4

The Manager_ID column in the table denotes recursive relationships in the employee data. There are two levels of managers.

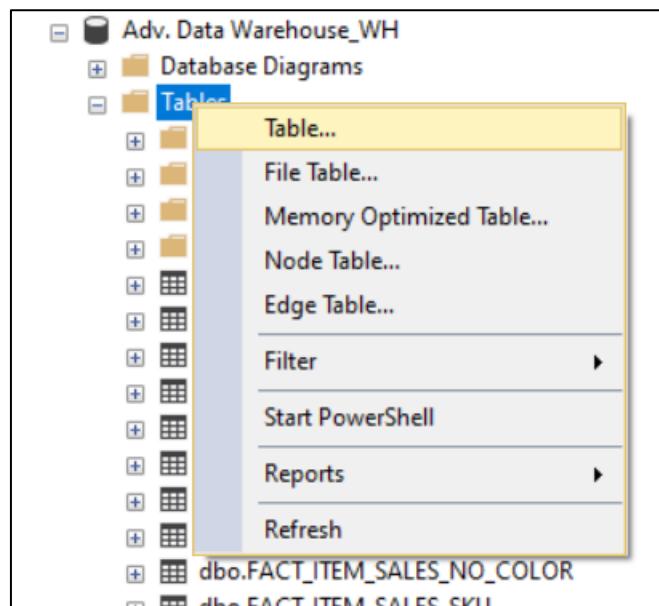
The lowest-level employees do not manage anyone. With this table in its current form, you cannot run a report that displays the relationships between employees. You need to flatten the LU_EMPLOYEE_RECURATION table to make such a report possible.

2. Close the table. Keep the database open.

Flatten the Lookup Table for Employees

Create the lookup table for the Level 1 manager

3. In the **Adv. Data Warehouse_WH** database, right click the folder name **Tables** and select **table**



4. This action displays an empty table.

In the table, create the following columns:

The image below also shows the appropriate data type to use for each column.

Column Name	Data Type
Level1_Mgr_ID	int
Level1_Mgr_Name	char(10)

5. Right-click the Level1_Mgr_ID column and select Primary Key.
6. Click Save.
7. In the Save As window, in the Table Name box, type LU_LEVEL1_MANAGER as the table name.
8. Click OK.
9. Enter the following record into the table:

Level1_Mgr_ID	Level1_Mgr_Name
1	Matt Wilson

10. Save and close the table.

Create the lookup table for Level 2 managers

1. In the **Adv. Data Warehouse_WH** database, right click the folder name **Tables** and select **table** In the table, create the following columns:

The image below also shows the appropriate data type to use for each column.

Column Name	Data Type
Level2_Mgr_ID	int
Level2_Mgr_Name	char(10)
Level1_Mgr_ID	int

2. Right-click the Level2_Mgr_ID column and select Primary Key.
3. Click Save.
4. In the Save As window, in the Table Name box, type LU_LEVEL2_MANAGER as the table name.
5. Click OK.
6. Enter the following records into the table:

Level2_Mgr_ID	Level2_Mgr_Name	Level1_Mgr_ID
1	Joe Thomburke	1
2	Mattie Nader	1
3	Tom Nelson	1

7. Save and close the table.

Create the lookup table for lowest-level employees

At this point, you would follow the steps above to create the lookup table for the lowest-level employees. However, this table has already been created for you.

1. In the Tables list, run the LU_EMPLOYEE_FLATTENED table.
The table data looks like the following:

Employee_ID	Employee_Name	Level2_Mgr_ID
1	Sandra Cooper	1
2	Jason Kidd	1
3	Jeff Tazewell	1
4	Tanya Cummins	1
5	Mary Sikes	1
6	Michael Ebert	1
7	Cassie Henderson	1
8	Jill Thompson	2
9	Nate Landers	2
10	Alex Snow	2
11	Ted Mills	2
12	Andrea Cosby	2
13	Miriam Green	2
14	Rachel Porter	2
15	Guy Fenn	2
16	James DeVille	2
17	Serena Williams	3
18	Amy Murphy	3
19	David Robbins	3

2. Close the table.

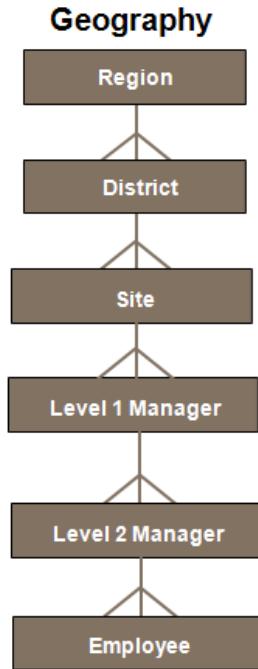
Configure the Recursive Hierarchies Project

Add the flattened lookup tables to the project and create the project attributes

1. In MicroStrategy Developer, in the Advanced Data Warehousing project source, open and select the Recursive Hierarchies Project.
2. On the Schema menu, select Architect to open the Architect Graphical Interface.
3. In the Architect graphical interface, on the Project Tables View tab, add the LU_LEVEL1_MANAGER, LU_LEVEL2_MANAGER, and LU_EMPLOYEE_FLATTENED tables to the project from the Advanced Data Warehousing Warehouse database instance.
 - In the Result Preview window, for each table, leave all the attributes selected and click OK.
4. Rename the Level1 Mgr attribute as Level 1 Manager and the Level2 Mgr attribute as Level 2 Manager. You need to create the Level 1 Manager, Level 2 Manager, and Employee attributes to map to the three levels of data in the recursive relationships.

Essentially, you are modifying the data model to look like the following:

Modified Logical Data Model for Recursive Hierarchies Project



In this exercise, you will create only the last three attributes in the hierarchy.

Create the attribute relationships

5. On the Hierarchy View tab, configure the parent-child relationships between the three attributes based on the logical data model above.
You need to click the parent attribute and drag the mouse pointer to the child attribute.
6. On the toolbar, click Save.
To make it easier to browse the project attributes, you need to create a user hierarchy.

Create a user hierarchy

7. In the Hierarchy View tab, create a user hierarchy named Geography that includes all three attributes. Make all three attributes entry points. Your user hierarchy should look like the following:



8. On the toolbar, click Save and Close.

Update the project schema

9. In the Schema Update window, click Update.

Move the Geography hierarchy to the Data Explorer folder

10. In the Schema objects folder, in the Hierarchies folder, move the Geography hierarchy to the Data Explorer folder.

This is required to be able to browse the Geography hierarchy in the Data Explorer.

Create and Run a Report with the Recursive Hierarchy

Create a report

1. In the Public Objects folder, in the Reports folder, create the following report:

Report View: 'Local Template'			Switch to:
Level 1 Manager	Level 2 Manager	Employee	

You should find the Level 1 Manager, Level 2 Manager and Employee attributes in the Geography hierarchy.

Run the report

2. Run the report. The result set should look like the following:

Level 1 Manager	Level 2 Manager	Employee
Matt Wilson	Joe Thornburke	Sandra Cooper
		Jason Kidd
		Jeff Tazewell
		Tanya Cummins
		Mary Sikes
		Michael Ebert
	Mattie Nader	Cassie Henderson
		Jill Thompson
		Nate Landers
Tom Nelson	Andrea Cosby	Alex Snow
		Ted Mills
		Miriam Green
		Rachel Porter
		Guy Fenn
		James DeVille
	Serena Williams	Serena Williams
		Amy Murphy
		David Robbins
		Gloria Brenner
		Andrew Radson
		Anna Radev

The report displays the three levels of employees in relationship to one another.

- Switch to the SQL View for the report. The SQL should look like the following:

```

select    a12.[Level1_Mgr_ID] AS Level1_Mgr_ID,
          a13.[Level1_Mgr_Name] AS Level1_Mgr_Name,
          a11.[Level2_Mgr_ID] AS Level2_Mgr_ID,
          a12.[Level2_Mgr_Name] AS Level2_Mgr_Name,
          a11.[Employee_ID] AS Employee_ID,
          a11.[Employee_Name] AS Employee_Name
from      [LU_EMPLOYEE_FLATTENED]      a11,
          [LU_LEVEL2_MANAGER]      a12,
          [LU_LEVEL1_MANAGER]      a13
where     a11.[Level2_Mgr_ID] = a12.[Level2_Mgr_ID] and
          a12.[Level1_Mgr_ID] = a13.[Level1_Mgr_ID]
  
```

The query retrieves the data from the three lookup tables that you created.

Save the report

- Switch to Grid View.
- Save the report in the Reports folder as Employee List and close the report.

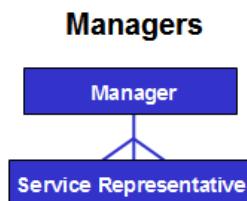
Using Logical Views - Recursion

During this exercise, you will create two logical views to display recursive relationships that exist between service representatives. Next, you will map these logical views to the appropriate attributes and create a report that displays the relationships between service representatives. You will perform these steps in the Recursive Hierarchies Project.

Recursion Information

This exercise is based on the following logical data model:

Logical Data Model



The Managers hierarchy looks like the following:

Managers Hierarchy



The schema for the project objects used in this exercise consists of the following table:

Schema

LU_CUSTOMER_SERVICE	
Service_Rep_ID	: Integer(4)
Service_Rep_Name	: NVarChar(50)
Manager_ID	: Integer(4)
Service_Center_ID	: Integer(4)
Service_Center_Name	: NVarChar(50)
Region_ID	: Integer(4)
Region_Name	: NVarChar(50)

The Manager_ID column in the LU_CUSTOMER_SERVICE table references the IDs in the Service_Rep_ID column. As such, this table contains a recursive relationship where specific service representatives

manage other service representatives. You need to create two logical views that serve as lookup tables for the Manager and Service Representative Attributes so that you can display this relationship in reports.

Recursion Exercise

Create the Logical View for the Manager Attribute

Open the Logical Table Editor

1. In the Schema Objects folder, in the Tables folder, right-click a blank area of the Object Viewer, point to New, and select Logical Table(W).

Define the logical view SQL

2. In the Logical View Exercises SQL.doc file, copy the SQL labeled as LVW_MANAGER and paste it in the SQL Statement pane of the Logical Table Editor. The SQL looks like the following:

```
Select Service_Rep_ID as Manager_ID,  
      Service_Rep_Name as Manager_Name  
From   LU_CUSTOMER_SERVICE  
Where  Manager_ID Is Null
```

Only service representatives who are managers have a null value for the Manager_ID column, so this query returns data only for those service representatives that qualify as managers.

3. Keep the Logical View Exercises SQL.doc file open as you will use it later in the exercises.

Define the logical view columns

4. In the Logical Table Editor, in the Object Browser, expand the LU_CUSTOMER_SERVICE table.
5. Drag the Manager_ID column to the Mapping pane.
6. In the Mapping pane, click Add.
7. In the Column Object box, type Manager_Name as the name of the column.
The column name must match the name used in the SQL statement.
8. In the Data Type drop-down list, select NVarChar as the data type for the column.
The completed column mapping should look like the following:

Column Object	Data Type	Pr...	S...
Manager_ID	Integer	4	
Manager_Name	NVarChar		

9. Save the logical table in the Tables folder as LVW_MANAGER and close the Logical Table Editor.

Create the Logical View for the Service Representative Attribute

Open the Logical Table Editor

1. In the Tables folder, right-click a blank area of the Object Viewer, point to New, and select Logical Table.

Define the logical view SQL

2. In the Logical View Exercises SQL.doc file, copy the SQL labeled as LVW_SERVICE_REPRESENTATIVE and paste it in the SQL Statement pane of the Logical Table Editor. The SQL looks like the following:

```

Select Service_Rep_ID, Service_Rep_Name,
      Manager_ID
From   LU_CUSTOMER_SERVICE
Where  Manager_ID Is Not Null

```

Only service representatives who are not managers have a value for the Manager_ID column, so this query returns data only for those service representatives who do not manage anyone else.

- Close the Logical View Exercises SQL.doc file.

Define the logical view columns

- In the Logical Table Editor, in the Object Browser, expand the LU_CUSTOMER_SERVICE table.
- Drag the Service_Rep_ID column to the Mapping pane.
- Drag the Service_Rep_Name column to the Mapping pane.
- Drag the Manager_ID column to the Mapping pane.

The completed column mapping should look like the following:

Column Object	Data Type	Pr...	S...
Service_Rep_ID	Integer	4	
Service_Rep_Name	NVarChar		
Manager_ID	Integer	4	

Save the logical view

- Save the logical table in the Tables folder as LVW_SERVICE_REPRESENTATIVE and close the Logical Table Editor.

Map Attributes to the Logical View

Map the existing attributes to the logical view columns

- In the Attributes folder, open the Service Representative attribute.
- Modify the ID form to select LVW_SERVICE_REPRESENTATIVE as the primary lookup table.
- Modify the DESC form to select LVW_SERVICE_REPRESENTATIVE as the primary lookup table.
- Save and close the Service Representative attribute.
- Open the Manager attribute.
- Modify the ID form to select LVW_MANAGER as the primary lookup table and LVW_Service_Representative as a source table.
- Create a DESC form that maps to the Manager_Name column in the LVW_MANAGER table.
- On the Children tab, select LVW_SERVICE_REPRESENTATIVE as the relationship table that is used to relate the Manager attribute to the Service Representative attribute.
- Remove the ID form from the list of report display forms and browse forms to ensure that only the manager name displays on reports and when browsing.
The ID form is automatically part of the default report display because it is the only form that existed when the attribute was originally created.
- Save and close the Manager attribute.

Update the project schema

11. Update the project schema.

Create a Report That Uses the Logical Views

Create a report

1. In the Public Objects folder, in the Reports folder, create the following report:

Report View: 'Local Template'		Switch to:
Manager	Service Representative	

You can access the attributes from the Managers hierarchy.

Run the report

2. Run the report. The result set should look like the following:

Manager	Service Representative
	Allen Green
Sara Adams	Hannah Campbell
	Terry Smith
Todd Whitmore	Eric Brinkman
	Melanie Jackson
Heather Moore	David Bell
	James Upton
	Daniel Davis
	Katrina Willis
Carissa Stone	Austin Jacobs
	Carol Stockton
	Grace Webb
	Thomas Brown
	Ronald Williams
Myra Miller	Isaiah McClure
	Ellen Dean
	Judy White
Jared Meyers	Peter Thornton
	Susan Fisher

Notice that the report displays each manager and the service representatives whom they manage.

3. Switch to the SQL View for the report. You should see the following SQL:

```

select a11.[Manager_ID] AS Manager_ID,
       a12.[Manager_Name] AS Manager_Name,
       a11.[Service_Rep_ID] AS Service_Rep_ID,
       a11.[Service_Rep_Name] AS Service_Rep_Name
  from (Select Service_Rep_ID, Service_Rep_Name,
              Manager_ID
        From LU_CUSTOMER_SERVICE
       Where Manager_ID Is Not Null) a11,
       (Select Service_Rep_ID as Manager_ID,
              Service_Rep_Name as Manager_Name
        From LU_CUSTOMER_SERVICE
       Where Manager_ID Is Null) a12
 where a11.[Manager_ID] = a12.[Manager_ID]

```

Notice that the FROM clause contains two derived table expressions where you would normally see physical table names. These expressions are the SQL statements for the logical views. The first one selects the necessary information from the LU_CUSTOMER_SERVICE table for the service representatives.

The second expression selects the necessary information from the LU_CUSTOMER_SERVICE table for the managers.

Save the report

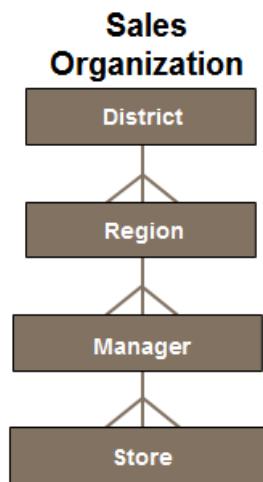
4. Switch to Grid View.
5. Save the report in the Reports folder as Service Representative Organization and close the report.

CHAPTER 15: SLOWLY CHANGING DIMENSIONS

Types of Slowly Changing Dimensions (SCDs)

Slowly Changing Dimensions (SCDs) refers to the process of tracking and analyzing attribute relationships that change over time. For example, a retail company has numerous stores that are assigned to various managers. The sales organization hierarchy in their data model is structured as follows:

Logical Data Model for Sales Organization Hierarchy



Although the relationships between districts, regions and managers do not change much over time, the managers assigned to stores change as they are reassigned or as they enter or leave the company.

Running a report to view the stores to which a manager is assigned is a standard query that looks at the current state of the data. Though, if the company wants to view reports that show past stores to which a manager has been assigned, this type of query requires a historical view of the data. So in this example the sales organization changes slowly in time as the managers are reorganized, that is, managers switch stores in time.

For example, the store lookup table and a fact table for the sales for each store contain the following data:

Lookup Table and Fact Table Data

LU_STORE for 10/12

Store_ID	Store_Name	Manager_ID	Manager_Name
...
13	Metro North	3	Missy Wells
17	Metro South	3	Missy Wells
20	Metro Central	2	Jim Smith
24	Metro East	2	Jim Smith
27	Metro West	2	Jim Smith
18	Long Branch	35	Jena Hill
...

FACT_STORE_SALES

Store_ID	Date_ID	Sales
...
13 (Metro North)	10/4/2012	10
17 (Metro South)	10/4/2012	15
20 (Metro Central)	10/4/2012	8
24 (Metro East)	10/4/2012	8
27 (Metro West)	10/4/2012	11
18 (Long Branch)	10/4/2012	9
...
13 (Metro North)	11/4/2012	13
17 (Metro South)	11/4/2012	10
20 (Metro Central)	11/4/2012	8
24 (Metro East)	11/4/2012	14
27 (Metro West)	11/4/2012	12
...

LU_STORE for 11/12

Store_ID	Store_Name	Manager_ID	Manager_Name
...
13	Metro North	3	Missy Wells
17	Metro South	31	Liz Townsend
20	Metro Central	2	Jim Smith
24	Metro East	2	Jim Smith
27	Metro West	3	Missy Wells
18	Long Branch	35	Jena Hill
...

Resulting Report?

Manager	Metrics	Sales		
		Month	October 2012	November 2012
Jena Hill			?	?
Jim Smith			?	?
Liz Townsend			?	?
Missy Wells			?	?

In November 2012, there was an organizational change as new managers were hired and existing ones were reassigned to different stores. The sales fact table records the daily sales for each store, but if you run a report like the sample report shown above, how are the October 2012 sales to be calculated for managers who changed stores? Are data for new managers to be included in the report? The answer depends on whether you are interested in analysing changes in data relationships across time.

Although SCDs are well documented in data warehousing literature; there are different terminologies used to distinguish the different types of SCDs. In this lesson we will discuss four types of SCDs, they are:

- As Is vs. As Is (also referred to as Type I SCDs)
- As Is vs. As Was (also referred to as Type II SCDs)
- Like vs. Like
- As Was vs. As Was

As Is vs. As Is (Type I SCDs)

As Is vs. As Is (Type I) involves analyzing all data in accordance with the attribute relationships as they exist currently. Regardless of how relationships have changed over time, you aggregate and qualify all data (current and historical) based on the current values in the lookup and relationship tables. If aggregate tables exist, you either have to modify how the values roll up to reflect the current attribute relationships, or you have to ignore the tables when you perform this type of analysis.

For example, using the sample store data, the LU_STORE table would be structured as follows:

Lookup Table for As Is vs. As Is (Type 1 SCDs)

LU_STORE

Store_ID	Store_Name	Manager_ID	Manager_Name
...
13	Metro North	3	Missy Wells
17	Metro South	31	Liz Townsend
20	Metro Central	2	Jim Smith
24	Metro East	2	Jim Smith
27	Metro West	3	Missy Wells
18	Long Branch	35	Jena Hill
...

The LU_STORE table records only the values for the attribute relationships as they exist currently.

Missy Wells is still associated with Metro North. She managed Metro South in 10/12 but not anymore.

Liz Townsend was hired in 11/12 and is assigned to manage Metro South.

Jim Smith managed Metro West in 10/12 but now he only has Metro Central and Metro West.

Missy Wells is now in charge of Metro West. Jena Hill did not get reassigned.

The LU_STORE table reflects the stores to which managers are currently assigned. This schema preserves only the current relationships in the LU_STORE table. As a result, when you run a report to aggregate the amount of sales for each manager, the fact table rolls up to the store to which the manager is currently assigned:

Report Result for As Is vs. As Is (Type I SCDs)

Store_ID	Manager_ID
...	...
13	3 (Missy Wells)
17	31 (Liz Townsend)
20	2 (Jim Smith)
24	2 (Jim Smith)
27	3 (Missy Wells)
18	35 (Jena Hill)
...	...

Store_ID	Date_ID	Sales
...
13 (Metro North)	10/4/2012	10
17 (Metro South)	10/4/2012	15
20 (Metro Central)	10/4/2012	8
24 (Metro East)	10/4/2012	8
27 (Metro West)	10/4/2012	11
18 (Long Branch)	10/4/2012	9
...
13 (Metro North)	11/4/2012	13
17 (Metro South)	11/4/2012	10
20 (Metro Central)	11/4/2012	8
24 (Metro East)	11/4/2012	14
27 (Metro West)	11/4/2012	12
18 (Long Branch)	11/4/2012	11
...

FACT_STORE_SALES

Missy = Metro North + Metro West

Liz = Metro South

Jim = Metro Central + Metro East

Jena = Long Branch

Resulting Report

Manager	Metrics	Sales		
		Month	October 2012	November 2012
Jena Hill			9	11
Jim Smith			16	22
Liz Townsend			15	10
Missy Wells			21	25

The illustration above displays part of the LU_STORE table showing the stores associated with a manager. The Manager_ID column is not in the FACT_STORE_SALES table, but this column is shown in several of the SCD illustrations in this lesson to make it easier to understand how store sales are rolled up to managers.

Notice that for Missy and Jim, the sales for October are only for the stores they manage as of November. For Liz even though she did not start as an employee until November, she has sales for October.

As Is vs. As Was (Type II SCDs)

As Is vs. As Was (Type II) involves analyzing all data in accordance with the attribute relationships as they exist currently and as they existed historically.

You aggregate and qualify data based on the values in the lookup and relationship tables that correspond to the desired timeframe. If aggregate tables exist, the logic behind how the values roll up may differ based on the time period that you query.

For example, using the same sample data, the LU_STORE table would be structured as follows:

Lookup Table for As Is vs. As Was (Type II SCDs)

LU_STORE

Store_ID	Store_Name	Manager_ID	Manager_Name	Start_Date	End_Date	Current_Flag
...
13	Metro North	3	Missy Wells	9/1/2011	12/31/2099	Y
17	Metro South	3	Missy Wells	2/1/2012	10/31/2012	N
20	Metro Central	2	Jim Smith	3/1/2009	12/31/2099	Y
24	Metro East	2	Jim Smith	6/1/2010	12/31/2099	Y
27	Metro West	2	Jim Smith	7/1/2011	10/31/2012	N
18	Long Branch	35	Jena Hill	5/1/2010	12/31/2099	Y
17	MetroSouth	31	Liz Townsend	11/1/2012	12/31/2099	Y
27	Metro West	3	Missy Wells	11/1/2012	12/31/2099	Y
...

The start and end dates comprise a life stamp that indicates the time period in which a manager was assigned a store.

The current flag column is a status flag that displays whether a row contains a current value (Y) or a historical value (N).

As the manager-store relationships change over time, there will be multiple records not only to map managers to their current stores, but also to map them to every store to which they were previously assigned. You can store these relationships in a single lookup table along with data range values and flags that indicate the time period when a particular relationship existed.

This schema preserves both the historical and current relationships. As a result, when you run a report to aggregate the total sales for each manager, each store record in the fact table rolls up to the manager who was assigned that store at the time that the sales occurred:

Report Result for As Is vs. As Was (Type II SCDs)

Store_ID	Manager_ID
...	...
13	3 (Missy Wells)
17	3 (Missy Wells)
20	2 (Jim Smith)
24	2 (Jim Smith)
27	2 (Jim Smith)
18	35 (Jena Hill)
...	...
13	3 (Missy Wells)
17	31 Liz Townsend
20	2 (Jim Smith)
24	2 (Jim Smith)
27	3 Missy Wells
18	35 (Jena Hill)
...	...

Store_ID	Date_ID	Sales
...
13 (Metro North)	10/4/2012	10
17 (Metro South)	10/4/2012	15
20 (Metro Central)	10/4/2012	8
24 (Metro East)	10/4/2012	8
27 (Metro West)	10/4/2012	11
18 (Long Branch)	10/4/2012	9
...
13 (Metro North)	11/4/2012	13
17 (Metro South)	11/4/2012	10
20 (Metro Central)	11/4/2012	8
24 (Metro East)	11/4/2012	14
27 (Metro West)	11/4/2012	12
18 (Long Branch)	11/4/2012	11
...

FACT_STORE_SALES

10/12 Missy = Metro North and South

10/12 Jim = Metro Central, East and West

10/12 Jena = Long Branch

11/12 Missy = Metro North and West

11/12 Jim = Metro Central and East

11/12 Jena = Long Branch

11/12 Liz = Metro South

Resulting Report

Manager	Metrics	Sales	
	Month	October 2012	November 2012
Jena Hill		9	11
Jim Smith		27	22
Liz Townsend		0	10
Missy Wells		25	25

Since Liz did not start as a manager until November, she does not have any sales number for October.

Like vs. Like

Like vs. Like, also referred to as comparable analysis, involves analyzing only data records that exist and are identical for the querying time period. In other words, only data relationships that have not changed over time are included in the result set.

Similar to the As Is vs. As Was or Type II SCDs, the schema preserves both the historical and current relationships. As a result, when you run a report to aggregate the total sales for each manager, only data that exists unchanged are part of the final result set:

Report Result for Like vs. Like

Store_ID	Manager_ID
...	...
13	3 (Missy Wells)
17	3 (Missy Wells)
20	2 (Jim Smith)
24	2 (Jim Smith)
27	2 (Jim Smith)
18	35 (Jena Hill)
...	...
13	3 (Missy Wells)
17	31 Liz Townsend
20	2 (Jim Smith)
24	2 (Jim Smith)
27	3 Missy Wells)
18	35 (Jena Hill)
...	...

Store_ID	Date_ID	Sales
...
13 (Metro North)	10/4/2012	10
17 (Metro South)	10/4/2012	15
20 (Metro Central)	10/4/2012	8
24 (Metro East)	10/4/2012	8
27 (Metro West)	10/4/2012	11
18 (Long Branch)	10/4/2012	9
...
13 (Metro North)	11/4/2012	13
17 (Metro South)	11/4/2012	10
20 (Metro Central)	11/4/2012	8
24 (Metro East)	11/4/2012	14
27 (Metro West)	11/4/2012	12
18 (Long Branch)	11/4/2012	11
...

FACT_STORE_SALES

10/12 Jena = Long Branch
10/12 Jim = Metro Central
an East

10/12 Missy = Metro North
11/12 Jena = Long Branch
11/12 Jim = Metro Central
an East

11/12 Missy = Metro North

Resulting Report

Manager	Metrics	Sales		
		Month	October 2012	November 2012
Jena Hill			9	11
Jim Smith			16	22
Missy Wells			10	13

Missy and Jim were reassigned stores in November and Liz came on as a new manager in November. Only Jena stayed identical in both time periods. Thus, the report contains sales from those stores that had the same managers in both time periods.

As Was vs. As Was

As Was vs. As Was involves analyzing data only in accordance with the attribute relationships as they existed historically. Similar to the As Is vs. As Was or Type II SCDs, the schema preserves both the historical and current relationships. With this type of analysis, however, you reference only historical relationships in queries. As a result, when you run a report to aggregate the sales for each manager, the fact table rolls up the store sales to which the manager was historically assigned, not the store that they are currently assigned:

Report Result for As Was vs. As Was

Store_ID	Manager_ID
...	...
13	3 (Missy Wells)
17	3 (Missy Wells)
20	2 (Jim Smith)
24	2 (Jim Smith)
27	2 (Jim Smith)
18	35 (Jena Hill)
...	...
13	3 (Missy Wells)
17	31 Liz Townsend
20	2 (Jim Smith)
24	2 (Jim Smith)
27	3 Missy Wells
18	35 (Jena Hill)
...	...

Store_ID	Date_ID	Sales
...
13 (Metro North)	10/4/2012	10
17 (Metro South)	10/4/2012	15
20 (Metro Central)	10/4/2012	8
24 (Metro East)	10/4/2012	8
27 (Metro West)	10/4/2012	11
18 (Long Branch)	10/4/2012	9
...
13 (Metro North)	11/4/2012	13
17 (Metro South)	11/4/2012	10
20 (Metro Central)	11/4/2012	8
24 (Metro East)	11/4/2012	14
27 (Metro West)	11/4/2012	12
18 (Long Branch)	11/4/2012	11
...

FACT_STORE_SALES

10/12 Jena = Long Branch

10/12 Jim = Metro Central, East and West

10/12 Missy = Metro North and South

11/12 Jena = Long Branch

11/12 Jim = Metro Central, East and West

11/12 Missy = Metro North and South

Resulting Report

Manager	Metrics	Sales		
		Month	October 2012	November 2012
Jena Hill			9	11
Jim Smith			27	34
Missy Wells			25	23

Missy's sales for both months roll up into the stores that she was initially assigned to even though in November she no longer managers Metro South and is instead responsible for Metro West. Similarly, Jim's sales for both months roll up into the three stores he was initially assigned to even though he no longer manages Metro West in November. Since Liz did not start as an employee until November, she is not included in the report. To include her

November sales for the Metro South store, you must query for data based on current relationships.

Summary of Four Types of SCDs

In summary, there are four types of SCDs:

- As Is vs. As Is (Type I)
- As Is vs. As Was (Type II)
- Like vs. Like
- As Was vs. As Was

Each of these types of analysis returns a different result set when querying the same data:

Report Results for Each Type of SCDs

As Is Vs. As Is (Type I)			As Is Vs. As Was (Type II)				
Only Current Data			Historical and Current Data				
	Metrics	Sales			Metrics	Sales	
	Month	October 2012	November 2012		Month	October 2012	November 2012
Manager				Manager			
Jena Hill		9	11	Jena Hill		9	11
Jim Smith		16	22	Jim Smith		27	22
Liz Townsend		15	10	Liz Townsend		0	10
Missy Wells		21	25	Missy Wells		25	25

Like Vs. Like			As Was Vs. As Was				
Intersection between Historical and Current Data			Only Historical Data				
	Metrics	Sales			Metrics	Sales	
	Month	October 2012	November 2012		Month	October 2012	November 2012
Manager				Manager			
Jena Hill		9	11	Jena Hill		9	11
Jim Smith		16	22	Jim Smith		27	34
Missy Wells		10	13	Missy Wells		25	23

As Is vs. As Is (Type I) analysis is the most common type of query performed since you do not preserve any history of attribute relationships. If data is time independent and users do not require historical comparisons, this type of analysis is sufficient to support a variety of reporting requirements.

When users do want to track changes in attribute relationships across time, As Is vs. As Was (Type II) and Like vs. Like analysis are the SCDs that are most frequently used. If data is time dependent and users are interested in analysing the changes in relationships (As Is vs. As Was or Type II) or in determining what relationships have remained constant (Like vs. Like), these types of analysis are required to fully support reporting requirements. Both of these types of analysis require a more complex data warehouse structure since you must do one of the following:

- Create columns in the affected lookup table to denote which values are current and which are historical
- Create multiple versions of lookup tables to store both current and historical values
- Modify the fact table structure to include the versioned relationship

In addition to having to maintain a more complex database design, the SQL generated is more complex and may take longer to process.

As Was vs. As Was analysis, in which users are interested only in tracking historical values, is rarely seen as a requirement. This type of analysis also requires the same changes to the data warehouse structure as As Is vs. As Was (Type II) or Like vs. Like analysis.

If reporting requirements include the need for SCDs, you can implement them using one of the following methods:

- Creating a life stamp (uses a single lookup table)

- Creating a hidden attribute that relates to both current and historical values (uses multiple lookup tables)
- Denormalizing the fact table (changes the fact table structure to accommodate SCDs)

Creating a Life Stamp

If a report requires time dependent analysis, one way to ensure that the query retrieves the appropriate data is to include the time period for which you want to view data in the SQL itself. You can then aggregate records according to the attribute relationships as they existed at that point in time. You can implement this solution by creating a life stamp.

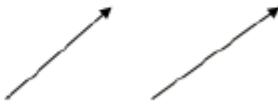
A life stamp consists of a start date and end date that indicate the time period for which specific records are valid. For example, in the sample scenario, you could modify the LU_STORE table as follows:

Lookup Table for Stores with Life Stamp

LU_STORE

Store_ID	Store_Name	Manager_ID	Manager_Name	Start_Date	End_Date
...
13	Metro North	3	Missy Wells	9/1/2011	12/31/2099
17	Metro South	3	Missy Wells	2/1/2012	10/31/2012
20	Metro Central	2	Jim Smith	3/1/2009	12/31/2099
24	Metro East	2	Jim Smith	6/1/2010	12/31/2099
27	Metro West	2	Jim Smith	7/1/2011	10/31/2012
18	Long Branch	35	Jena Hill	5/1/2010	12/31/2099
17	MetroSouth	31	Liz Townsend	11/1/2012	12/31/2099
27	Metro West	3	Missy Wells	11/1/2012	12/31/2099
...

The start and end dates comprise a life stamp that indicates the time period in which a manager was assigned a store.



Using this method, the LU_STORE table functions as a single lookup table that contains records not only to map managers to their current stores, but also to map them to every store to which they have been previously assigned. By using start and end dates, you can determine the validity of any particular record from the record itself. For records that represent the current store assignments, the end date is set arbitrarily large (in this example, 12/31/2099).

If you implement versioning using life stamps, you need to do the following:

1. Modify the lookup table to include start date and end date columns.
2. Create a Start Date attribute and map it to the start date column in the lookup table.
3. Create an End Date attribute and map it to the end date column in the lookup table.

4. Make the Start Date and End Date both parents of the Store attribute with a one-to-many relationship.
5. Include the desired date range in the report filter.

Based on the date range that you include in the report filter, the report aggregates data according to the stores that the managers were responsible for during the specified time period. Essentially, the life stamp determines which version of the manager-store relationship is used for aggregation.

For example, in the sample scenario, you could use the following filter to achieve As Is vs. As Was (Type II) analysis:

Report Result Using a Life Stamp

Store_ID	Store_Name	Manager_ID	Manager_Name	Start_Date	End_Date
...
13	Metro North	3	Missy Wells	9/1/2011	12/31/2099
17	Metro South	3	Missy Wells	2/1/2012	10/31/2012
20	Metro Central	2	Jim Smith	3/1/2009	12/31/2099
24	Metro East	2	Jim Smith	6/1/2010	12/31/2099
27	Metro West	2	Jim Smith	7/1/2011	10/31/2012
18	Long Branch	35	Jena Hill	5/1/2010	12/31/2099
17	MetroSouth	31	Liz Townsend	11/1/2012	12/31/2099
27	Metro West	3	Missy Wells	11/1/2012	12/31/2099

Store_ID	Date_ID	Sales
13 (Metro North)	10/4/2012	10
17 (Metro South)	10/4/2012	15
20 (Metro Central)	10/4/2012	8
24 (Metro East)	10/4/2012	8
27 (Metro West)	10/4/2012	11
18 (Long Branch)	10/4/2012	9
...
13 (Metro North)	11/4/2012	13
17 (Metro South)	11/4/2012	10
20 (Metro Central)	11/4/2012	8
24 (Metro East)	11/4/2012	14
27 (Metro West)	11/4/2012	12
18 (Long Branch)	11/4/2012	11

LU_STORE

As Is Vs. As Was

Date >= Start Date

and

Date < End Date

and

Month in (10/12, 11/12)

FACT_STORE_SALES

Resulting Report

Manager	Metrics	Sales		
		Month	October 2012	November 2012
Jena Hill			9	11
Jim Smith			27	22
Liz Townsend			0	10
Missy Wells			25	25

Using the date range in the filter, the SQL Engine joins the start and end dates to the dates in the fact table to retrieve the records for the requested time period. The report aggregates the sales based on the stores to which managers were assigned during that timeframe. The SQL for the report looks like the following:

```

select a12.[Manager_ID] AS Manager_ID,
max(a12.[Manager_Name]) AS Manager_Name,
a13.[Month_ID] AS Month_ID,
max(a13.Month_Desc) AS Month_Desc,
sum(a11.[Sales])AS WJXBFS1
from [FACT_STORE _SALES] a11,

```

```

[LU_STORE] a12,
[LU_DATE] a13,
where a11.[STORE_ID] = a12.[STORE_ID]
and a11.[Date_ID] = a13.[Date_ID]
and (a11.[Date_ID] >= a12.[Start_Date]
and a11.[Date_ID] < a12.[End_Date]
and a13.[Month_ID] in (201210, 201211))
group by a12.[Manager_ID], a13.[Month_ID]

```

When you use life stamps to implement SCDs, over time, a single lookup table can store many different historical versions of the data, along with the current definition. For queries in which you simply want to view the current data, filtering using the lifestamp can be a tedious, time-consuming method of retrieving the most current data from the lookup table.

You can make it easier to retrieve the rows that reflect the current attribute relationships by using a status flag to indicate which records in a table are historical and which are current. For example, in the sample scenario, you could modify the LU_STORE table as follows:

Lookup Table for Stores with Current Flag

LU_STORE

Store_ID	Store_Name	Manager_ID	Manager_Name	Start_Date	End_Date	Current_Flag
...
13	Metro North	3	Missy Wells	9/1/2011	12/31/2099	Y
17	Metro South	3	Missy Wells	2/1/2012	10/31/2012	N
20	Metro Central	2	Jim Smith	3/1/2009	12/31/2099	Y
24	Metro East	2	Jim Smith	6/1/2010	12/31/2099	Y
27	Metro West	2	Jim Smith	7/1/2011	10/31/2012	N
18	Long Branch	35	Jena Hill	5/1/2010	12/31/2099	Y
17	MetroSouth	31	Liz Townsend	11/1/2012	12/31/2099	Y
27	Metro West	3	Missy Wells	11/1/2012	12/31/2099	Y
...

The Current_Flag column is a status flag that displays whether a row contains a current value (Y) or a historical value (N).

The Current_Flag column in the LU_STORE table indicates whether each record in the table contains a current value (Y) or historical value (N). You can easily perform As Is vs. As Is (Type I) analysis on this table by simply filtering on the Current Flag being set to “Y.”

Using a Hidden Attribute for SCDs

Now, you will learn how to use hidden attributes to implement SCDs. You will have two separate tables—one to store just the current information and one to store current and historical information. You can then create a third lookup table that stores a record for every manager for every store that they were assigned. You use

this table to join the values in fact tables to either the current or historical versions of the lookup tables as needed. You then create a hidden attribute to reference this lookup table.

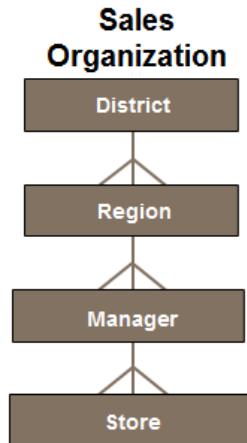
If you implement SCDs using a hidden attribute, you need to do the following:

1. Modify the data model to include current and historical attributes that represent the different “versions” of the data.
2. Create separate lookup tables for the versioned attributes to store current and historical values.
3. Create attributes to map to the current and historical versions of the lookup tables.
4. Create a separate lookup table that contains a record for every manager for every store to which they were assigned.
5. Create a hidden attribute that maps to this lookup table and relate it to both the current and historical attributes.
6. Key affected fact tables based on the ID of the hidden attribute.

Modifying the Logical Data Model

In the sample scenario, the original data model for the Sales Organization hierarchy looks like the following:

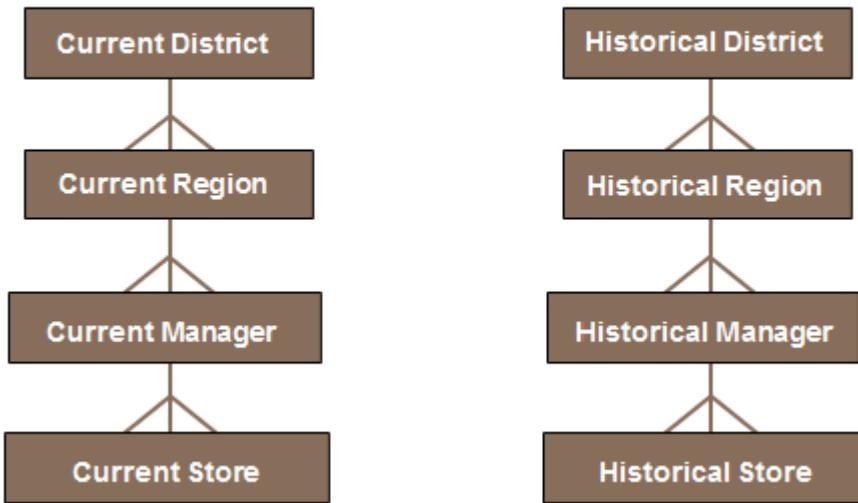
Logical Data Model for Sales Organization Hierarchy



Supporting different versions of an attribute entails logically separating the attributes that reference current and historical information. Therefore, you need to modify the Sales Organization hierarchy to include both current and historical attributes as follows:

Modified Logical Data Model for Sales Organization Hierarchy

Sales Organization



The modified Sales Organization hierarchy contains a branch for the current version of each attribute and a branch for the historical version of each attribute. Right now, the data model does not show a join between the two branches. Later, you will use the hidden attribute to relate the two branches, enabling you to join fact table data to either current or historical manager information.

Creating Current and Historical Versions of the Lookup Tables

After modifying the data model for the Sales Organization hierarchy, you need to create current and historical versions of the lookup tables in the data warehouse so that you have tables to which you can map the current and historical attributes. The lookup table for the Current Manager attribute looks like the following:

Lookup Table for Current Manager Attribute

LU_CURR_STORE

Curr_Store_ID	Curr_Store_Name	Curr_Manager_ID	Curr_Manager_Name
...
13	Metro North	3	Missy Wells
20	Metro Central	2	Jim Smith
24	Metro East	2	Jim Smith
18	Long Branch	35	Jena Hill
17	MetroSouth	31	Liz Townsend
27	Metro West	3	Missy Wells
...

LU_CURR_STORE
maintains only the
current information
for managers.

The LU_CURR_STORE table stores only the most current information for managers. In this table, each manager is related only to the stores to which they are currently assigned.

The lookup table for the Historical Store attribute looks like the following:

Lookup Table for Historical Store Attribute

LU_HIST_STORE

Hist_Store_SPK_ID	Hist_Store_ID	Hist_Store_Name	Hist_Manager_ID	Hist_Manager_Name	MRR_Flag
...
1	13	Metro North	3	Missy Wells	Y
2	17	Metro South	3	Missy Wells	N
3	20	Metro Central	2	Jim Smith	Y
4	24	Metro East	2	Jim Smith	Y
5	27	Metro West	2	Jim Smith	N
6	18	Long Branch	35	Jena Hill	Y
7	17	MetroSouth	31	Liz Townsend	Y
8	27	Metro West	3	Missy Wells	Y
...

Hist_Store_SPK_ID is a surrogate key that functions as the primary key for the table. The Hist_Store_ID column cannot be the primary key since the same store will have multiple records in this table as different managers are assigned over time.

LU_HIST_STORE maintains the historical and current information for managers.

The LU_HIST_STORE table stores both the current and historical information for managers. In this table, each manager is related to every store to which they were previously assigned. The Hist_Store_SPK_ID column serves as a surrogate primary key. The Hist_Store_ID is not sufficient as the primary key since the same store will have multiple records in the table because of different managers are assigned over time. Thus, the surrogate key is necessary to uniquely identify each row in the table. For example, the Metro West store has two records, one for Jim who was the previous manager and one for Missy who is the current manager.

Both of the illustrations only show the lookup tables that will eventually map to the Current Store, Current Manager, Historical Store and Historical Manager attributes. The higher-level attributes in both branches of the Sales Organization hierarchy (for example, Current Region and Historical Region) also require tables to which you can map their attribute definitions.

If you have a distinct list of current versus historical regions, you need to build separate lookup tables. At each attribute level, you would have both historical and current versions of the lookup tables. However, if Region does not change over time then Current Region and Historical Region attributes can pull from the same list of regions. You do not need separate lookup tables. You can treat them as role attributes, and you can map them using table views, automatic attribute role recognition, or explicit table aliasing.

Creating Current and Historical Attributes

After you create the necessary lookup tables, you are ready to create the current and historical attributes from the modified data model.

When you create the Current Store attribute, its ID form maps to the Curr_Store_ID column and its DESC form to the Curr_Store_Name column in the LU_CURR_STORE table. When you create the Current Manager attribute, its ID form maps to the Curr_Manager_ID column and its DESC form to the Curr_Manager_Name column in the LU_CURR_STORE table:

Mapping of Current Store and Manager Attributes

LU_CURR_STORE



Curr_Store_ID	Curr_Store_Name	Curr_Manager_ID	Curr_Manager_Name
...
13	Metro North	3	Missy Wells
20	Metro Central	2	Jim Smith
24	Metro East	2	Jim Smith
18	Long Branch	35	Jena Hill
17	MetroSouth	31	Liz Townsend
27	Metro West	3	Missy Wells
...

The Current Store attribute maps to the Curr_Store_ID and Curr_Store_Name columns. The Current Manager attribute maps to the Curr_Manager_ID and Curr_Manager_Name in the LU_CURR_STORE table.

When you create the Historical Store attribute, its ID form maps to the Hist_Store_SPK_ID column and its DESC form to the Hist_Store_Name column in the LU_HIST_STORE table. When you create the Historical Manager attribute, its ID form maps to the Hist_Manager_ID column and its DESC form to the Hist_Manager_Name column in the LU_HIST_STORE table:

Mapping of Historical Store and Manager Attributes

LU_HIST_STORE



Hist_Store_SPK_ID	Hist_Store_ID	Hist_Store_Name	Hist_Manager_ID	Hist_Manager_Name	MRR_Flag
...
1	13	Metro North	3	Missy Wells	Y
2	17	Metro South	3	Missy Wells	N
3	20	Metro Central	2	Jim Smith	Y
4	24	Metro East	2	Jim Smith	Y
5	27	Metro West	2	Jim Smith	N
6	18	Long Branch	35	Jena Hill	Y
7	17	MetroSouth	31	Liz Townsend	Y
8	27	Metro West	3	Missy Wells	Y
...

The Historical Store attribute maps to the Hist_Store_SPK_ID and Hist_Store_Name columns. The Historical Manager attribute maps to the Hist_Manager_ID and Hist_Manager_Name columns in the LU_HIST_STORE table.

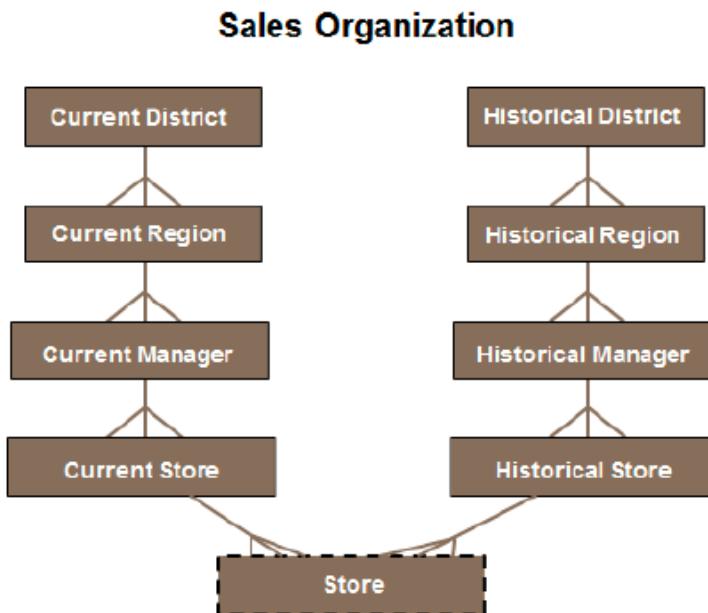
You also need to create the other higher-level attributes in both branches and map them to their respective lookup tables. After creating the higher-level attributes, you need to define the parent-child relationships for both branches.

Creating the Hidden Attribute

After creating the current and historical lookup tables and defining the current and historical attributes, you are ready to create the hidden attribute that you will use to create a join path from either branch of the hierarchy to the fact tables.

To set up the hidden attribute, you first need to modify the logical data model for the Sales Organization hierarchy to look like the following:

Modified Logical Data Model for Sales Organization Hierarchy with the Hidden Attribute



The Store attribute ties together the two branches of the hierarchy and enables you to join either branch to fact table data. It exists only to provide a consolidated join path to the fact tables. It is not logically relevant to users, so it should not be visible to them. In reports, users will see the Current Store and Historical Store attributes, depending on the type of analysis they want to perform. These two objects comprise the logical representation of stores that users see. In the background, the Store attribute, which is a hidden attribute, will join elements from either the LU_CURR_STORE or LU_HIST_STORE table to the relevant fact data.

To set up the hidden attribute, you need to create a third lookup table that looks like the following:

Lookup Table for the Store Hidden Attribute

LU_STORE

Store_SPK_ID	Curr_Store_ID	Hist_Store_SPK_ID

1	13 (Metro North)	1 (Metro North)
2		2 (Metro South)
3	20 (Metro Central)	3 (Metro Central)
4	24 (Metro East)	4 (Metro East)
5		5 (Metro West)
6	18 (Long Branch)	6 (Long Branch)
7	17 (Metro South)	7 (Metro South)
8	27 (Metro West)	8 (Metro West)
...

Store_SPK_ID is a surrogate key that functions as the primary key for the table since the same store will have multiple records in this table if there has been more than one manager assigned.

LU_STORE stores a record for every store that has ever been assigned to a manager.

The Curr_Store_ID column relates this table to the LU_CURR_STORE table, while the Hist_Store_SPK_ID column relates this table to the LU_HIST_STORE table.

The LU_STORE table contains a record for every manager who has been assigned one or more stores throughout their employment with the organization. Since a store appears more than once in this table if the store has been managed by more than one manager, the table contains a surrogate primary key, Store_SPK_ID. The Curr_Store_ID column relates records in this table to the LU_CURR_STORE table, and the Hist_Emp_SPK_ID column relates them to the LU_HIST_STORE. Because it contains a foreign key to both the historical and current versions of store information, this table provides a join path between fact tables and either version of the lookup table.

After creating this LU_STORE table, you need to do the following:

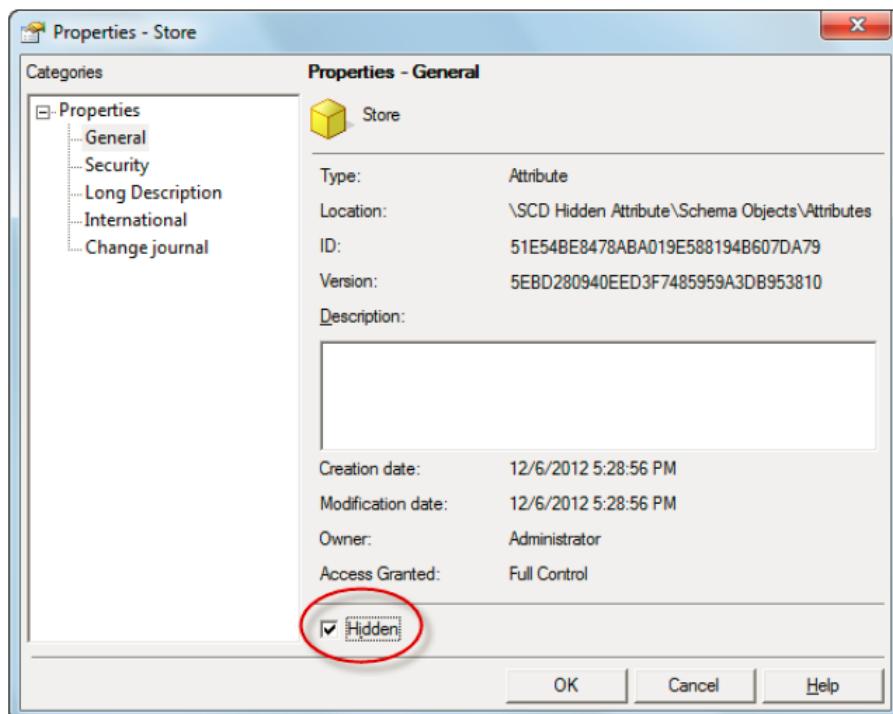
1. Create the Store attribute, mapping it to the Store_SPK_ID column in the LU_STORE table.
2. Make the Store attribute a child of both the Current Store and Historical Store attributes (one-to-many relationship to both attributes).
3. Make the Store attribute a hidden attribute.

To create a hidden attribute:

1. Expand the Schema Objects folder.
2. In the Schema Objects folder, select the Attributes folder.
3. In the Attributes folder, right-click the attribute you want to hide and select Properties.
4. In the Properties window, in the Categories list, under the General category, select the Hidden check box.
5. Click OK.

The following image shows the option for making an object hidden:

Hidden Object Option



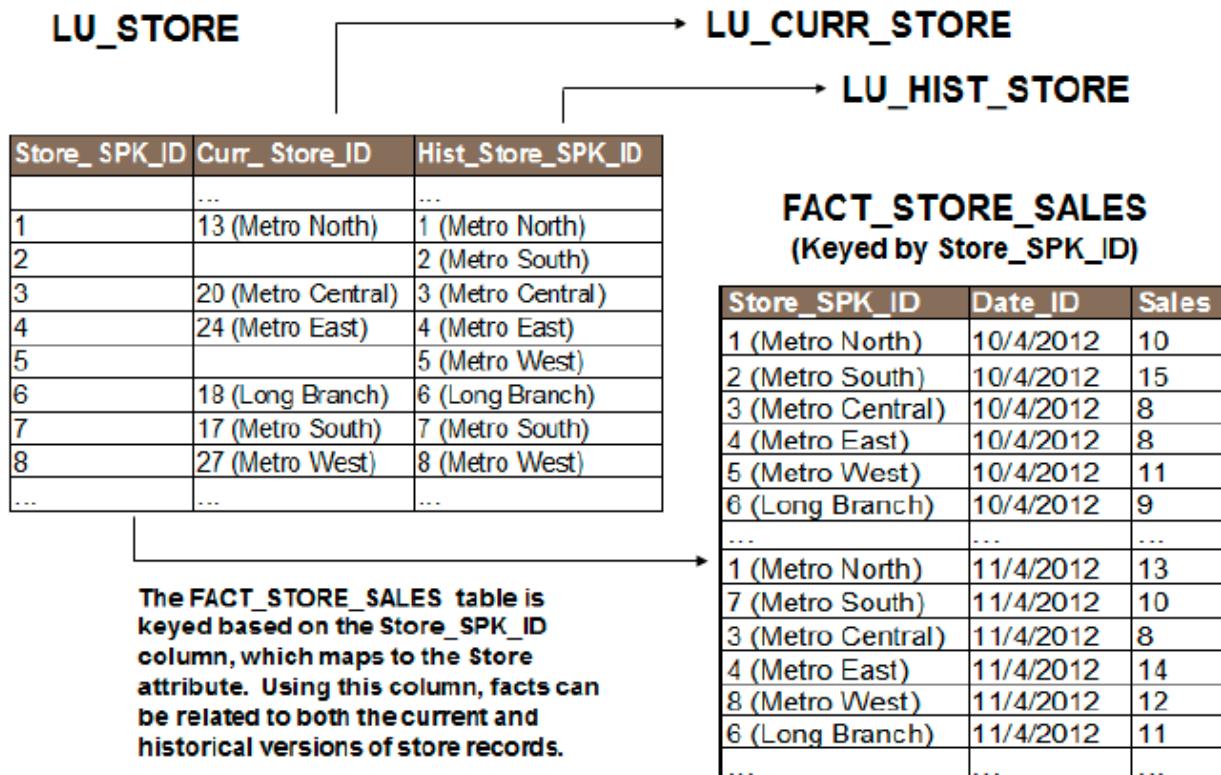
Creating the Store attribute as a hidden attribute makes it available to use in the join path, while hiding it from the view of users to eliminate confusion. Users do not need to directly access the Store attribute since it is never used as a display attribute on reports.

Keying Fact Tables Based on the Hidden Attribute

The last step in implementing a hidden attribute to handle SCDs is to key fact tables based on the ID column of the hidden attribute. For any facts you want to analyze for current or historical employee information, you need to ensure that the fact tables are keyed using the `Store_SPK_ID` column, the surrogate primary key in the `LU_STORE` table. Doing so ensures that joins from the fact tables to either the historical or current data always occur through the Store attribute, which references both branches of the hierarchy.

The rekeyed `FACT_STORE_SALES` table looks like the following:

Fact Table Keyed on Hidden Attribute



Because the FACT_STORE_SALES table is now keyed based on the Store attribute, which relates to both the Current Store and Historical Store attributes, you can join from the fact table to either “version” of store information. You can join the Hist_Store_SPK_ID column in the LU_STORE table to the same column in the LU_HIST_STORE table to retrieve current or historical records for stores and relate them to facts. You can join the Curr_Store_ID column in the LU_STORE table to the same column in the LU_CURR_STORE table to retrieve current records for stores and relate them to facts.

Analysis Using a Hidden Attribute

With the hidden attribute solution in place, you can perform the various types of analyses.

For example, if you want to sales just for stores to which managers are currently assigned, you could run the following report:

Report Result Set with Current Manager-Store Information

Current Manager	Current Store	Metrics	Sales
Jim Smith	Metro Central		16
	Metro East		22
Missy Wells	Metro North		23
	Metro West		12
Liz Townsend	Metro South		10
Jena Hill	Long Branch		20

Since you want to view current information, the template contains the Current Manager and Store attributes, which ensures that the result set is retrieved from the LU_CURR_STORE table. The SQL for this report looks like the following:

```
select a13.[Curr_Manager_ID] AS Curr_Manager_ID,  
max(a13.[Curr_Manager_Name]) AS Curr_Manager_Name,  
a12.[Curr_Store_ID] AS Curr_Store_ID,  
max(a13.[Curr_Store_Name]) AS Curr_Store_Name,  
sum(a11.[Sales]) AS WJXBFS1  
from [FACT_STORE_SALES] a11,  
[LU_STORE] a12,  
[LU_CURR_STORE] a13  
where a11.[Store_SPK_ID] = a12.[Store_SPK_ID] and  
a12.[Curr_Store_ID] = a13.[Curr_Store_ID]  
group by a13.[Curr_Manager_ID],  
a12.[Curr_Store_ID]
```

Notice that the LU_STORE table is in the FROM clause. In the WHERE clause, it joins to the fact table based on the Store_SPK_ID column, which maps to the hidden Store attribute. The SQL Engine then joins to the current version of the lookup table to retrieve the managers and their associated stores.

If you want to view just historical information for managers who have previously managed other stores, you could run the following report:

Report Result Set with Historical Manager-Store Information

Report with Historical Manager-Store Information

Report details

Report Filter:
(MRR Flag) = N

Historical Manager	Historical Store	Metrics	Sales
Jim Smith	Metro West		11
Missy Wells	Metro South		15

Filtering on the MRR Flag (Most Recent Record Flag) enables you to retrieve only historical records from the LU_HIST_STORE table.

Since you want to view historical information, the template contains the Historical Manager and Store attributes, which ensures that the result set is retrieved from the LU_HIST_STORE table. However, the

LU_HIST_STORE table contains both current and historical information (for performing current-historical comparisons), so you need to ensure that you only retrieve records from this table that are historical store assignments for managers. You limit the result set to the historical records by filtering on the MRR_Flag column in the LU_HIST_STORE table:

Lookup Table for Historical Store Information with MRR Flag

LU_HIST_STORE

Hist_Store_SPK_ID	Hist_Store_ID	Hist_Store_Name	Hist_Manager_ID	Hist_Manager_Name	MRR_Flag
...
1	13	Metro North	3	Missy Wells	Y
2	17	Metro South	3	Missy Wells	N
3	20	Metro Central	2	Jim Smith	Y
4	24	Metro East	2	Jim Smith	Y
5	27	Metro West	2	Jim Smith	N
6	18	Long Branch	35	Jena Hill	Y
7	17	MetroSouth	31	Liz Townsend	Y
8	27	Metro West	3	Missy Wells	Y
...

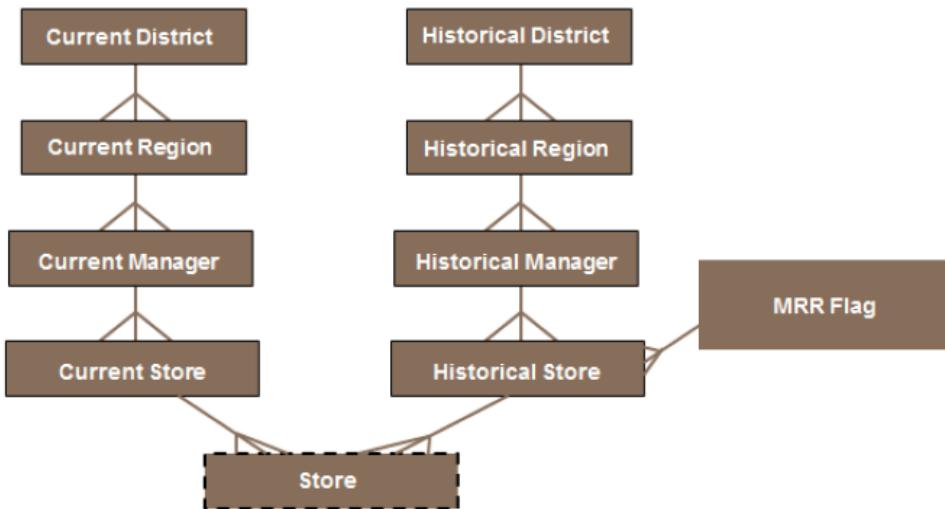
The MRR_Flag column indicates which records in the LU_HIST_STORE table are current manager-store assignments. Current assignments have a value of "Y," while past assignments have a value of "N."

The MRR_Flag column (MRR stands for most recent record) exists in the LU_HIST_STORE table to denote which records are current store assignments. Current assignments have a value of "Y," while past assignments have a value of "N."

To filter on this column, you need to create a MRR Flag attribute that maps to the MRR_Flag column in the LU_HIST_STORE table. This attribute is a parent of the Historical Store attribute. When you create the MRR Flag attribute, you include it in the logical data model as follows:

Logical Data Model for Sales Organization Hierarchy with MRR Flag

Sales Organization



After you have created this attribute, you can use it in the report filter to include only records where the MRR Flag is set to "N." This filter limits the result set to historical records. The SQL for this report looks like the following:

```

select a13.[Hist_Manager_ID] AS Hist_Manager_ID,
       max(a13.Hist_Manager_Name) AS Hist_Manager_Name,
       a12.[Hist_Store_SPK_ID] AS Hist_Store_SPK_ID,
       max(a13.[Hist_Store_Name]) AS Hist_Store_Name,
       sum(a11.[Sales]) AS WJXBFS1
  from [FACT_STORE_SALES] a11,
       [LU_STORE] a12,
       [LU_HIST_STORE] a13,
 where a11.[Store_SPK_ID] = a12.[Store_SPK_ID] and
       a12.[Hist_Store_SPK_ID] =
       a13.[Hist_Store_SPK_ID] and
       a13.[MRR_Flag] in ('N')
 group by a13.[Hist_Manager_ID],
       a12.[Hist_Store_SPK_ID]
  
```

Notice that the LU_STORE table is in the FROM clause. In the WHERE clause, it joins to the fact table based on the Store_SPK_ID column, which maps to the hidden Employee attribute. The SQL Engine then joins to the historical version of the lookup table to retrieve the stores and their associated managers.

Though the WHERE clause includes a condition that filters on the MRR_Flag column to retrieve only records for which the MRR_Flag value is "N". Because of this filter, the SQL Engine retrieves only historical records from the table.

If you want to view both current and historical information for managers, you could run the following report:

Report Result Set with Current and Historical Manager-Store Information

MRR Flag	Historical Manager	Historical Store	Metrics	Sales
N	Jim Smith	Metro West		11
	Missy Wells	Metro South		15
Y	Jim Smith	Metro Central	16	
		Metro East	22	
	Missy Wells	Metro North	23	
		Metro West	12	
	Liz Townsend	Metro South	10	
	Jena Hill	Long Branch		20

Since you want to view current and historical information, the query needs to access the LU_HIST_STORE table, which contains both current and historical store assignments for each manager. Therefore, the template contains the Historical Manager and Store attributes. If the MRR Flag is also used on the template, then you can further distinguish between the current and historical records. The SQL for this report looks like the following:

```

select a13.[Hist_Manager_ID] AS Hist_Manager_ID,
max(a13.[Hist_Manager_Name]) AS Hist_Manager_Name,
a12.[Hist_Store_SPK_ID] AS Hist_Store_SPK_ID,
max(a13.[Hist_Store_Name]) AS Hist_Store_Name,
a13.[MRR_Flag] AS MRR_Flag,
sum(a11.[Sales]) AS WJXBFS1
from [FACT_STORE_SALES] a11,
[LU_STORE] a12,
[LU_HIST_STORE] a13
where a11.[Store_SPK_ID] = a12.[Store_SPK_ID] and
a12.[Hist_Store_SPK_ID] = a13.[Hist_Store_SPK_ID]
group by a13.[Hist_Manager_ID],
a12.[Hist_Store_SPK_ID],
a13.[MRR_Flag]
```

Notice that the LU_STORE table is in the FROM clause. In the WHERE clause, it joins to the fact table based on the Store_SPK_ID column, which maps to the hidden Employee attribute. The SQL Engine then joins to the historical version of the lookup table to retrieve the managers and their associated stores.

Because no filter exists, the SQL Engine retrieves all of the manager records from the table, both current and historical store assignments.

These reports provide a few examples of SCDs analysis using a hidden attribute. Using more advanced reporting functionality, like conditional metrics, you can build even more complex reports to compare historical sales to current sales.

[Analysis of All Store Sales](#)

The hidden attribute solution provides a logical and physical separation between current and historical data that makes it easy for users to understand.

By defining both historical and current attributes in the Sales Organization hierarchy, users can easily place specific attributes on a report to view the sales generated by the various managers for their current and previous stores. What if your users also want to be able to run a report that displays the total sales by each store regardless of the current or previous manager for a store?

The LU_HIST_STORE table contains records that associate stores with all of their managers, both current and historical. However, if you run a report that displays only the Historical Store and Sales, the result looks like the following:

Report Result Set with Historical Store Attribute

Report Result Set with Historical Store Only (All Sales)

The report displays multiple rows for Metro South and West since these two stores have multiple records in the LU_HIST_STORE table.

Historical Store	Metrics	Sales
Metro North		23
Metro South		15
Metro South		10
Metro Central		16
Metro East		22
Metro West		11
Metro West		12
Long Branch		20

Hist_Store_SPK_ID	Hist_Store_ID	Hist_Store_Name	Hist_Manager_ID	Hist_Manager_Name	MRR_Flag
...
1	13	Metro North	3	Missy Wells	Y
2	17	Metro South	3	Missy Wells	N
3	20	Metro Central	2	Jim Smith	Y
4	24	Metro East	2	Jim Smith	Y
5	27	Metro West	2	Jim Smith	N
6	18	Long Branch	35	Jena Hill	Y
7	17	MetroSouth	31	Liz Townsend	Y
8	27	Metro West	3	Missy Wells	Y
...

The report displays multiple records for Metro South and West in the result set because multiple records exist for them in the LU_HIST_STORE table. The LU_HIST_STORE table stores a record for each store that has been managed by one or more managers. As a result, given the structure of this table, the Historical Store attribute cannot group the sales into a single row for each store in the report display. The same table structure that enables the attribute to accommodate “versions” prevents a simple grouping by store.

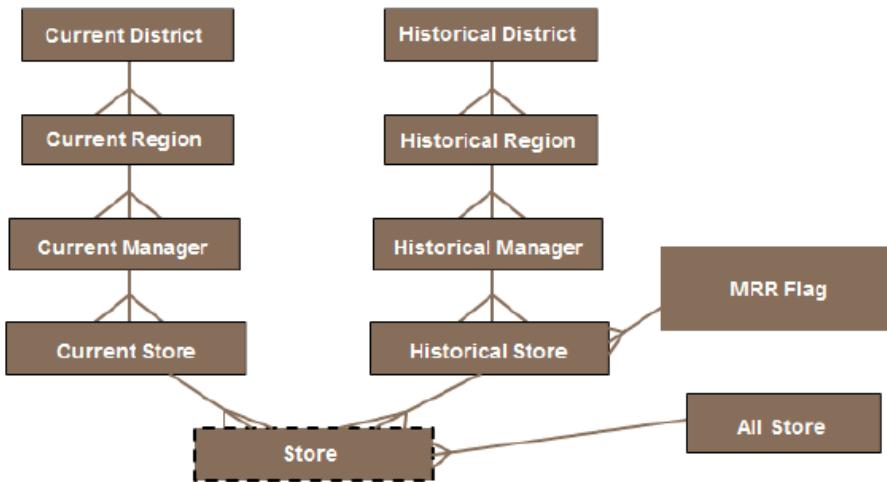
If you want to display the sales by each store regardless of the manager, you can modify the structure of the Sales Organization hierarchy and underlying tables to achieve such a report. To resolve this requirement, you need to do the following:

1. Modify the Sales Organization hierarchy to include a new attribute that relates to all of the records for a store, historical or current.
2. Modify the Sales Organization schema to accommodate the new attribute.
3. Create the new attribute.

First, you can modify the Sales Organization hierarchy to look like the following:

Modified Sales Organization Hierarchy

Sales Organization



Since the LU_STORE table contains all of the records for each store, you can use the Store attribute to join to the fact table and get the total sales for any store regardless of the manager. The All Store attribute relates all of the records for a single store, so that you can group them together in a report result set.

After modifying the hierarchy, you need to create a lookup table for the All Store attribute and modify the LU_STORE table so that you can relate the All Store and Store attributes. The LU_ALL_STORE and LU_STORE tables look like the following:

Lookup Table for All Store Attribute

LU_ALL_STORE

	All_Store_ID	All_Store_Name
...
1	1	Metro North
2	2	Metro South
3	3	Metro East
4	4	Metro West
5	5	Metro Central
6	6	Long Branch
...

The LU_ALL_STORE table contains one record for each store, so it can be used to group sales for each store regardless of the manager. The All_Store_ID column is added to the LU_STORE table to relate it to the LU_ALL_STORE table.

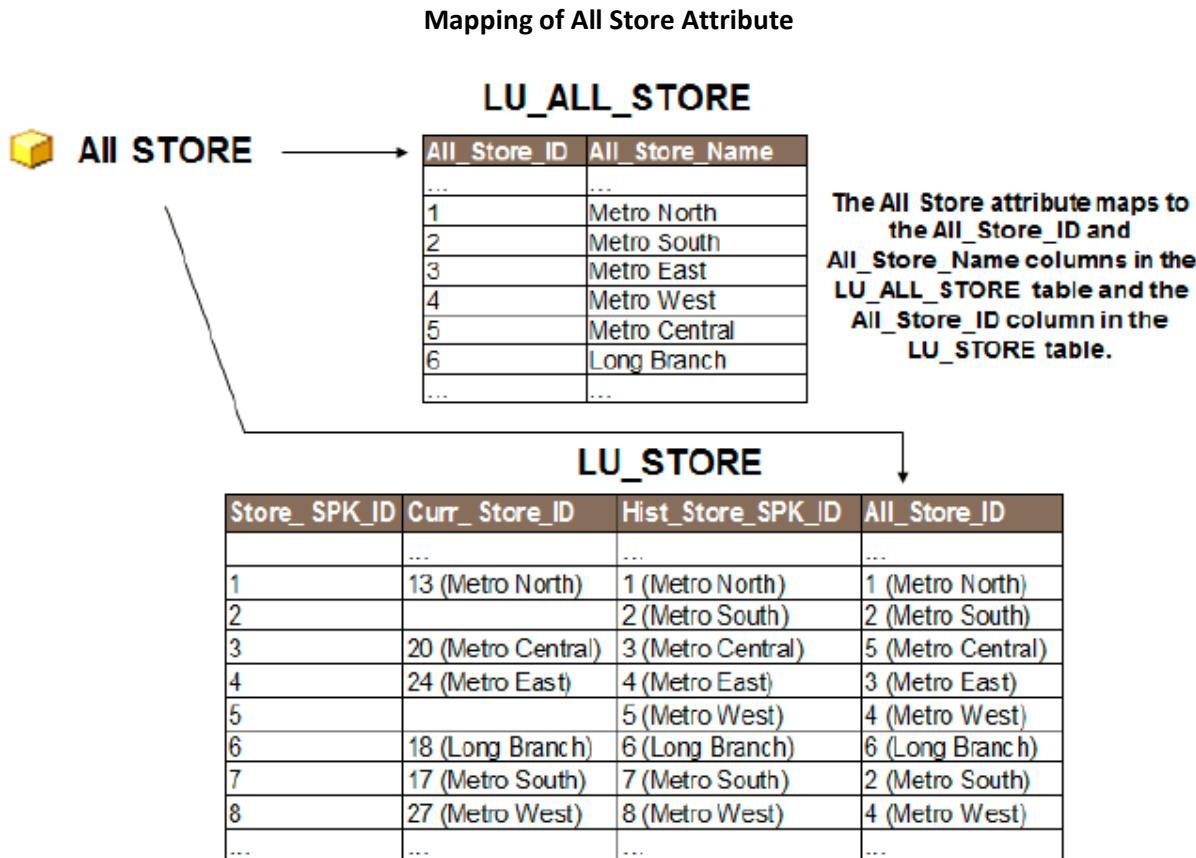
LU_STORE

Store_SPK_ID	Curr_Store_ID	Hist_Store_SPK_ID	All_Store_ID
...
1	13 (Metro North)	1 (Metro North)	1 (Metro North)
2		2 (Metro South)	2 (Metro South)
3	20 (Metro Central)	3 (Metro Central)	5 (Metro Central)
4	24 (Metro East)	4 (Metro East)	3 (Metro East)
5		5 (Metro West)	4 (Metro West)
6	18 (Long Branch)	6 (Long Branch)	6 (Long Branch)
7	17 (Metro South)	7 (Metro South)	2 (Metro South)
8	27 (Metro West)	8 (Metro West)	4 (Metro West)
...

The LU_ALL_STORE table contains a single record for each store. Therefore, you can use it as the lookup table from which to pull store information if you want to group sales by each store, regardless of manager. You modify the LU_STORE table to include the ID column from the LU_ALL_STORE table.

This foreign key relates the two tables.

After modifying the hierarchy and schema, you can create the All Store attribute and relate it to the hidden Store attribute as follows:



The All Store attribute maps to the All_Store_ID and All_Store_Name columns in the LU_ALL_STORE table as well as the All_Store_ID column in the LU_STORE table. You need to add the hidden Store attribute as a child of the All Store attribute with a one-to-many relationship.

Now, if you want to view the total sales for each store regardless of the managers, you can build a report that looks like the following:

Report Result Set with Total Sales by Store

Report Result Set with All Sales by Store

The report displays overall sales for each store regardless of the manager.

All Store	Metrics	Sales
Metro North		23
Metro South		25
Metro East		22
Metro West		23
Metro Central		16
Long Branch		20

The report displays the total sales by each store regardless of which managers were responsible for the store. The SQL for this report looks like the following:

```
select a12.[All_Store_ID] AS All_Store_ID,
       max(a13.[All_Store_Name]) AS All_Store_Name,
       sum(a11.[Sales]) AS WJXBFS1
  from FACT_STORE_SALES a11,
       [LU_STORE] a12,
       [LU_ALL_STORE] a13
 where a11.Store_SPK_ID = a12.Store_SPK_ID and
       a12.All_Store_ID = a13.All_Store_ID
 group by a12.All_Store_ID
```

Notice that the LU_STORE and LU_ALL_STORE tables are in the FROM clause. In the WHERE clause, the query joins to the fact table based on the Store_SPK_ID column, which maps to the hidden Store attribute. Then, it joins the Store attribute to the All Store attribute based on the All_Store_ID column that relates the two attributes.

Denormalizing Fact Tables

A final alternative for implementing SCDs is to denormalize the fact tables that contain the versioned attribute. Denormalization means introducing redundancy into how data is stored.

For example, the original structure of the FACT_STORE_SALES table looks like the following:

Original Fact Table Structure

FACT_STORE_SALES

Only the store and date IDs are contained in the fact table, so the sales data is stored only by store and date.

Store_ID	Date_ID	Sales
...
13 (Metro North)	10/4/2012	10
17 (Metro South)	10/4/2012	15
20 (Metro Central)	10/4/2012	8
24 (Metro East)	10/4/2012	8
27 (Metro West)	10/4/2012	11
18 (Long Branch)	10/4/2012	9
...
13 (Metro North)	11/4/2012	13
17 (Metro South)	11/4/2012	10
20 (Metro Central)	11/4/2012	8
24 (Metro East)	11/4/2012	14
27 (Metro West)	11/4/2012	12
18 (Long Branch)	11/4/2012	11
...

The FACT_STORE_SALES table stores only the store and date IDs, so the sales data is available only by store and date. Performing “version” analysis using this table is difficult because it does not contain manager information.

Since a manager can be assigned to different stores at different times, you have no way of knowing which store is associated with which manager on a given date.

You can denormalize the fact table to include not only the lowest level attribute (in this case, Store), but also the higher-level attribute (in this case, Manager as follows:

Denormalized Fact Table Structure

**FACT_STORE_SALES
(Denormalized)**

The manager, store, and date IDs are contained in the fact table, so the sales is stored by manager, store, and date.

Manager_ID	Store_ID	Date_ID	Sales
...
3 (Missy Wells)	13 (Metro North)	10/4/2012	10
3 (Missy Wells)	17 (Metro South)	10/4/2012	15
2 (Jim Smith)	20 (Metro Central)	10/4/2012	8
2 (Jim Smith)	24 (Metro East)	10/4/2012	8
2 (Jim Smith)	27 (Metro West)	10/4/2012	11
35 (Jena Hill)	18 (Long Branch)	10/4/2012	9
...
3 (Missy Wells)	13 (Metro North)	11/4/2012	13
31 (Liz Townsend)	17 (Metro South)	11/4/2012	10
2 (Jim Smith)	20 (Metro Central)	11/4/2012	8
2 (Jim Smith)	24 (Metro East)	11/4/2012	14
3 (Missy Wells)	27 (Metro West)	11/4/2012	11
35 (Jena Hill)	18 (Long Branch)	11/4/2012	12
...

With both the manager and store IDs present in the fact table, you have moved the relationship in which the changes occur into the fact table itself. You are now capturing the sales by manager, store, and date. Storing

the fact at the manager and store levels removes the need to incorporate SCDs into the lookup tables themselves.

The fact table is larger if manager information is stored in it, but this denormalization does provide a less complex answer to SCDs. Depending on the volume of data in your fact tables and the way in which data is captured in the source system, this option may or may not be viable in your own environment.

Exercises:

You should complete the following exercise in the Advanced Data Warehousing project source. Instructions at the beginning of each section reference the projects that you need to use for that particular exercise.

SCDs Overview

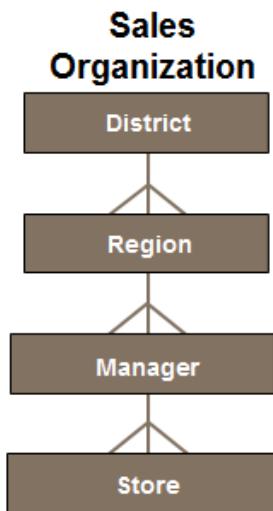
For the exercises in this section, you will build two projects; one that implements SCDs with life stamp and another that implements “versioned” attributes. You will create these projects in the Advanced Data Warehousing project source using tables provided in the data warehouse database. These tables enable you to set up the life stamps and hidden attribute solutions for SCDs. The data models and schemas for these projects are included with the exercises. You need to review this information before beginning the exercises.

During the exercises, you will first create the projects based on the data models and schemas provided. Next, you will create and run reports in the projects that use either life stamps or “versioned” attributes.

SCDs - Life Stamp Project Information

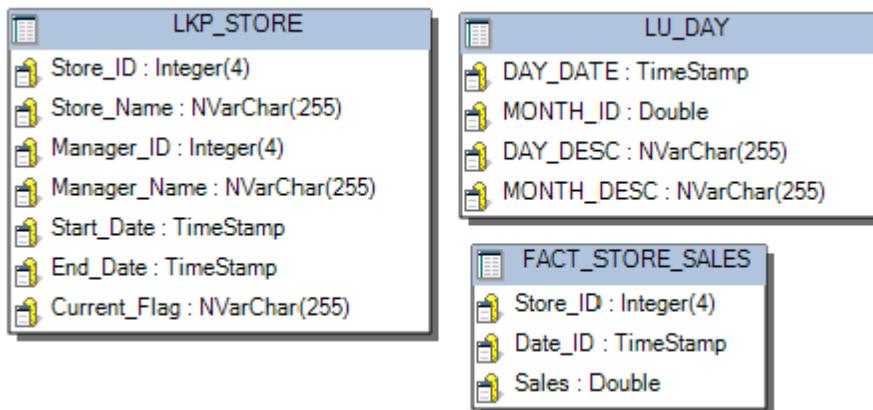
You should create the SCDs-Life Stamp Project based on the following logical data model:

SCDs - Life Stamp Logical Data Model



The schema for this project consists of the following three tables:

SCDs - Life Stamp Schema



These are the tables that you will need to add to the project from the ADVDW_WH database.

The LKP_STORE table data stores Stores and Managers information. The LU_DAY table serves as the look up table for both Day and Month attributes.

The FACT_STORE_SALES contains the daily sales information for a store.

Create the SCDs - Life Stamp Project

Create the project

1. In the Advanced Data Warehousing project source, open the Project Creation Assistant to create a new project.
2. Name the new project SCDs - Life Stamp Project.
3. Uncheck the Enable Change Journal for this project and Enable Quick Search for this project check boxes.
4. Click OK.

It takes a few minutes for MicroStrategy Architect to build the initial project files.

5. Click OK to skip the rest of the steps in the Project Creation Assistant.
6. In the pop-up information dialog box, click OK.
7. Open the SCDs - Life Stamp Project.
8. On the Schema menu, select Architect to open the Architect Graphical Interface.
9. In the Warehouse Database Instance window, select the Advanced Data Warehousing Warehouse database instance and click OK.

Add the tables to the project and create the project attributes

10. In the Architect graphical interface, on the Project Tables View tab, add the following tables, shown in the schema, to the project:
 - LKP_STORE
In the Result Preview window, keep all the attributes selected and click OK.
 - LU_DAY
In the Result Preview window, keep all the attributes selected and click OK.
 - FACT_STORE_SALES
In the Result Preview window, keep only the Store attribute selected and click OK

Modify the Day Date attribute

11. In the Properties pane, click the Attributes tab.
12. On the Attributes tab, in the drop-down list, select the Day attribute.
13. Modify the ID form by adding a second form expression that maps to the Date_ID column in the FACT_STORE_SALES table. Set the mapping method to Automatic.
14. Right-click the Day attribute and rename it Date.
15. In the Properties pane for the Date attribute, select Form1:ID and in the Datetime drop-down list select Date.

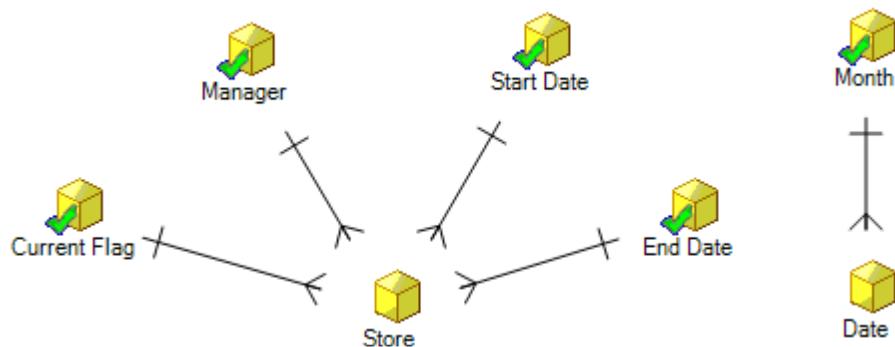
Change the format of Start Date and End Date attributes

16. On the Attributes tab, in the drop-down list, select the End Date attribute.
17. In the Properties pane for the End Date attribute, select Form1:ID and in the Datetime drop-down list select Date.
18. In the Attributes tab, in the drop-down list, select the Start Date attribute.
19. In the Properties pane for the Start Date attribute, select Form1:ID and in the Datetime drop-down list select Date.

Create the Current Flag attribute

20. On the Project Tables View tab, in the LKP_STORE table, select the CURRENT_FLAG column, right-click and select Create Attributes.
21. On the Hierarchy View tab, set up the one-to-many parent-child relationship between the Month and Date attributes. Repeat the step for Manager, Start Date, End Date, Current Flag and the Store attributes. You need to click the parent attribute and drag the mouse pointer to the child attribute.

Your system hierarchy should look like the following:

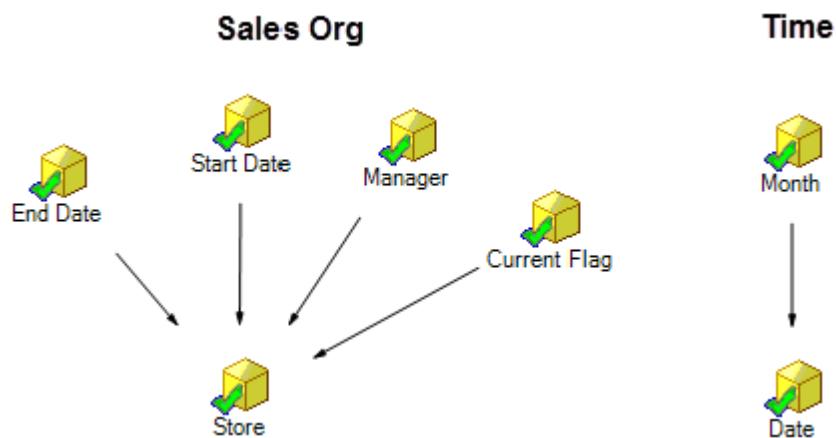


To make it easier for you to browse attributes when you build reports, you need to create user hierarchies.

Create user hierarchies

22. On the Hierarchy View tab, create a user hierarchy named Sales Org that includes the Current Flag, Manager, Start Date, End Date and Store attributes. Make all of the attributes entry points.
23. Create a second user hierarchy named Time that includes the Month and Date attributes. Make both attributes entry points.

Your user hierarchies should look like the following:



24. On the toolbar, click Save and Update Schema.
25. Close Architect.
26. In the Schema Update window, click Cancel.

Move the Time and Sales Org hierarchies to the Data Explorer folder

27 In Developer, in the Schema objects folder, in the Hierarchies folder, move the Time and Sales Org hierarchies to the Data Explorer folder.

This is required to be able to browse these hierarchies in the Data Explorer.

Create and Run Reports in the SCDs - Life Stamp Project

1. In the Public Objects folder, in the Metrics folder, rename the Sum of Sales metric as Sales. This metric was created automatically by the Architect Graphical Interface.

Create a report that shows As Is vs As Is (Type I) SCD information

2. In the Reports folder, create the following report:



You can find all the attributes in the Attributes folder in the Schema Objects folder. The Sales metric is in the Metrics folder.

Run the report

3. Run the report. The result set should look like the following:

Manager	Store		Month	October 2012	November 2012
			Metrics	Sales	Sales
Jim Smith	Metro Central			8	8
	Metro East			8	14
Missy Wells	Metro North			10	13
	Metro West			11	12
Liz Townsend	Metro South			15	10
Jena Hill	Long Branch			9	11

This report displays the October and November sales for each store and the manager who is currently assigned to a particular store.

4. Switch to the SQL View for the report. The SQL should look like the following:

```

select    a13.[Manager_ID] AS Manager_ID,
          max(a13.[Manager_Name]) AS Manager_Name,
          a11.[Store_ID] AS Store_ID,
          max(a13.[Store_Name]) AS Store_Name,
          a12.[MONTH_ID] AS MONTH_ID,
          max(a12.[MONTH_DESC]) AS MONTH_DESC,
          sum(a11.[Sales]) AS WJXBFS1
from      [FACT_STORE_SALES]      a11,
          [LU_DAY]           a12,
          [LKP_STORE]        a13
where     a11.[Date_ID] = a12.[DAY_DATE] and
          a11.[Store_ID] = a13.[Store_ID]
          and
          (a12.[MONTH_ID] in (201210, 201211)
          and a13.[Current_Flag] in ('Y'))
group by  a13.[Manager_ID],
          a11.[Store_ID],
          a12.[MONTH_ID]

```

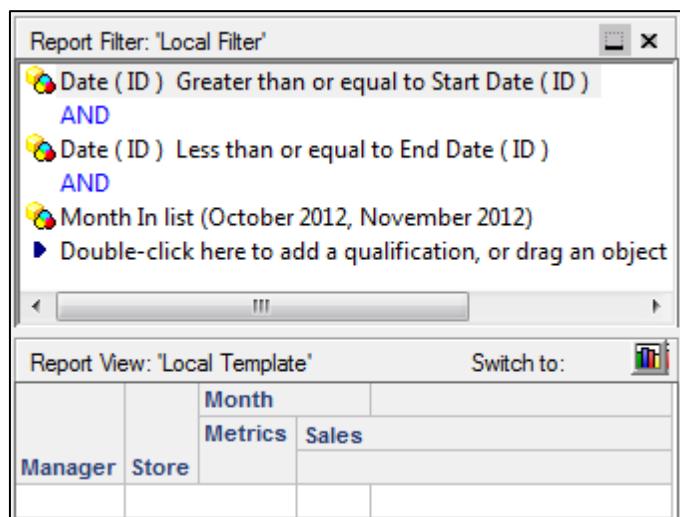
Notice that the query uses the Current Flag attribute to determine which records are current to obtain this result set.

Save the report

5. Switch to Grid View.
6. Save the report in the Reports folder as As Is vs As Is (Type I) and close the report.

Create a report with As Is vs As Was (Type II SCD) information

7. In the Reports folder, create the following report:



You use the Start Date and End Date attributes in the report filter to get the current and historical relationships between stores and managers.

Run the report

8. Run the report. The result set should look like the following:

Manager	Store	Month	October	November
			2012	2012
		Metrics	Sales	Sales
Jim Smith	Metro Central		8	8
	Metro East		8	14
	Metro West		11	
Missy Wells	Metro North		10	13
	Metro South		15	
	Metro West			12
Liz Townsend	Metro South			10
Jena Hill	Long Branch		9	11

This report displays the October and November Sales for each store.

It also shows who the current and previous managers are for a particular store. Notice how Metro West and Metro South stores display twice in the report. This is because different managers have been assigned to these stores during the reporting time period.

9. Switch to the SQL View for the report. The SQL should look like the following:

```

select    a12.[Manager_ID] AS Manager_ID,
          max(a12.[Manager_Name]) AS Manager_Name,
          a11.[Store_ID] AS Store_ID,
          max(a12.[Store_Name]) AS Store_Name,
          a13.[MONTH_ID] AS MONTH_ID,
          max(a13.[MONTH_DESC]) AS MONTH_DESC,
          sum(a11.[Sales]) AS WJXBFS1
from      [FACT_STORE_SALES]      a11,
          [LKP_STORE]        a12,
          [LU_DAY]           a13
where     a11.[Store_ID] = a12.[Store_ID] and
          a11.[Date_ID] = a13.[DAY_DATE]
and       (a11.[Date_ID] >= a12.[Start_Date]
          and a11.[Date_ID] <= a12.[End_Date])
          and a13.[MONTH_ID] in (201210, 201211))
group by  a12.[Manager_ID],
          a11.[Store_ID],
          a13.[MONTH_ID]

```

Notice how the query uses the Start Date and End Date attributes to aggregate the sales data.

Save the report

10. Switch to Grid View.
11. Save the report in the Reports folder as As Is vs As Was (Type II) and close the report.

Extra Challenge

Now you will create the next report on your own. The desired result set is shown at the end of the exercise.

Report Requirements

In the Reports folder, create a report that displays the sales for those stores that have not changed managers during the months of October and November.

In other words, create a report that displays Like vs Like analysis. After you run the report and view the result set, save the report as Like vs Like Analysis.

You need to create a report filter using the Start Date and End Date attributes.

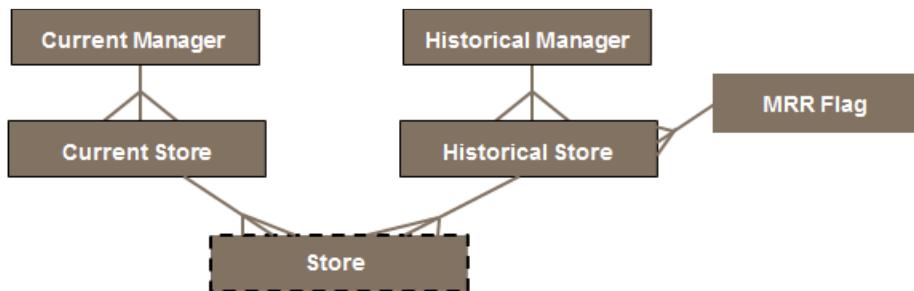
Report Result

Manager	Store	Month	October	November
			2012	2012
	Metrics	Sales	Sales	
Jim Smith	Metro Central		8	8
	Metro East		8	14
Missy Wells	Metro North		10	13
Jena Hill	Long Branch		9	11

SCDs - Hidden Attribute Project Information

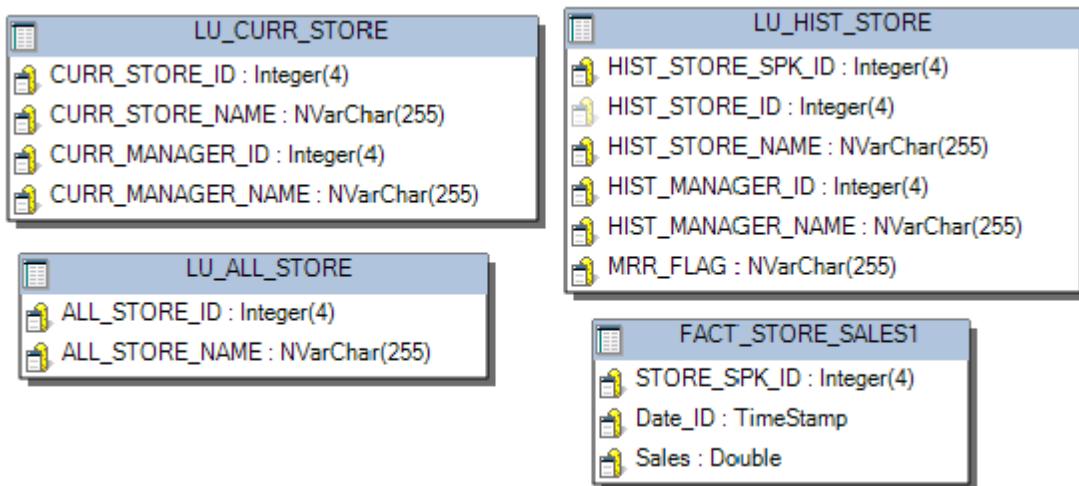
You should complete the following exercises using the Hidden Attribute Project found in the Advanced Data Warehousing project source. The hierarchies and schema for this project are included with the exercises. You need to review this information before beginning the exercises.

SCDs-Hidden Attribute Logical Data Model



The schema for this project consists of the following four tables:

SCDs-Hidden Attribute Schema



The LU_CURR_STORE table stores only current store and manager information. The LU_HIST_STORE table stores both current and historical store and manager information. The LU_STORE table stores a record every time a store is assigned to a different manager. This table is used to tie both the current and historical attributes to the fact table data.

Most of the attributes that you need have already been created in the project.

In the exercise below you will build the rest of the data model and perform analysis using the hidden attribute method.

SCDs - Hidden Attribute Exercises

Modify the Historical Store attribute

1. Open the SCDs - Hidden Attribute Project in Architect.
2. Create a DESC form for the Historical Store attribute, that maps to the HIST_STORE_NAME column in the LU_HIST_STORE table.
3. For the ID form, ensure that Use as Browse Form and Use as Report Form are set to False.

Create the Most Recent Record Flag attribute

4. On the Project Tables View tab, in the LU_HIST_STORE table, select the MRR_FLAG column, right-click and select Create Attributes.
5. Rename the newly created attribute MRR Flag.
6. On the Hierarchy View tab, set up the relationship between the MRR Flag and Historical Store attribute as shown in the logical data model.

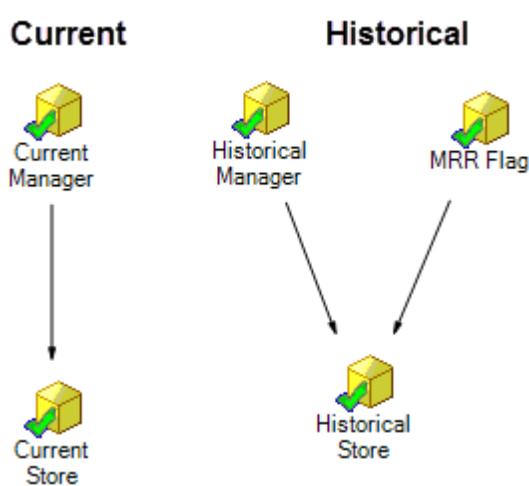
Hide the Store attribute

7. In the Properties pane, on the Attributes tab, in the drop-down list, select the Store attribute.
8. Under Hidden, select True.

To make it easier for you to browse attributes when you build reports, you need to create user hierarchies.

Create user hierarchies

9. On the Hierarchy View tab, create a user hierarchy named Current that includes the Current Store and Current Manager attributes. Make both attributes entry points.
 10. Create another hierarchy named Historical that includes the Historical Store, Historical Manager and MRR Flag attributes. Make all attributes entry points.
- Your two user hierarchies should look like the following:



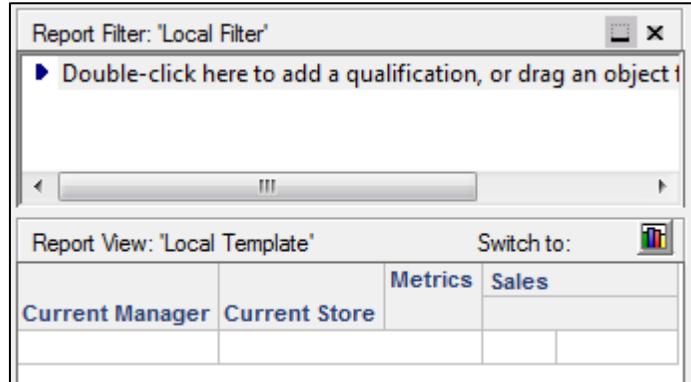
11. On the toolbar, click Save and Update Schema.
12. Close Architect.
13. In the Schema Update window, click Cancel.

Move the hierarchies to the Data Explorer folder

14. In Developer, in the Schema objects folder, in the Hierarchies folder, move the Current and Historical hierarchies to the Data Explorer folder.
This is required to be able to browse the hierarchies in the Data Explorer.

Create and Run Reports in the SCDs- Hidden Attribute Project

1. In the Public Objects folder, in the Metrics folder, rename the Sum of Stores Sales metric as Sales.
This metric was created automatically by the Architect Graphical Interface.
Create a report with current manager and store information
2. In the Reports folder, create the following report:
—



You can find the Current Manager and the Current Store attributes in the Attributes folder under the Schema Objects folder. The Sales metric is in the Metrics folder.

Run the report

- Run the report. The result set should look like the following:

Current Manager	Current Store	Metrics	Sales
Jim Smith	Metro Central		16
	Metro East		22
Missy Wells	Metro North		23
	Metro West		12
Liz Townsend	Metro South		10
Jena Hill	Long Branch		20

This report displays the sales for each manager and the store(s) to which they are currently assigned.

- Switch to the SQL View for the report. The SQL should look like the following:

```

select    a13.[CURR_MANAGER_ID] AS CURR_MANAGER_ID,
          max(a13.[CURR_MANAGER_NAME]) AS CURR_MANAGER_NAME,
          a12.[CURR_STORE_ID] AS CURR_STORE_ID,
          max(a13.[CURR_STORE_NAME]) AS CURR_STORE_NAME,
          sum(a11.[Sales]) AS WJXBFS1
from      [FACT_STORE_SALES1]      a11,
          [LU_STORE]           a12,
          [LU_CURR_STORE]       a13
where     a11.[STORE_SPK_ID] = a12.[STORE_SPK_ID] and
          a12.[CURR_STORE_ID] = a13.[CURR_STORE_ID]
group by  a13.[CURR_MANAGER_ID],
          a12.[CURR_STORE_ID]
  
```

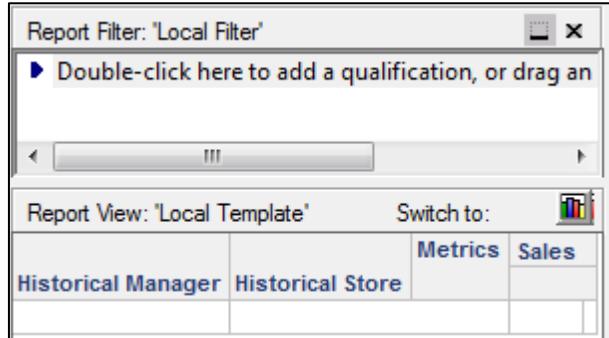
The query accesses the LU_CURR_STORE table to obtain this result set.

Save the report

- Switch to Grid View.
- Save the report in the Reports folder as Current Manager Information and close the report.

Create a report with current and historical store manager information

- In the Reports folder, create the following report:



You use the historical attributes because both current and historical information is stored in the LU_HIST_STORE table.

Run the report

- Run the report. The result set should look like the following:

Historical Manager	Historical Store	Metrics	Sales
Jim Smith	Metro Central		16
	Metro East		22
	Metro West		11
Missy Wells	Metro North		23
	Metro South		15
	Metro West		12
Liz Townsend	Metro South		10
Jena Hill	Long Branch		20

This report displays the sales for stores to which managers were assigned previously and to those they are currently responsible.

Notice that Metro West and Metro South display twice in the report because these stores have had different managers over the reporting period.

- Switch to the SQL View for the report. The SQL should look like the following:

```

select    a13.[HIST_MANAGER_ID] AS HIST_MANAGER_ID,
          max(a13.[HIST_MANAGER_NAME]) AS HIST_MANAGER_NAME,
          a12.[HIST_STORE_SPK_ID] AS HIST_STORE_SPK_ID,
          max(a13.[HIST_STORE_NAME]) AS HIST_STORE_NAME,
          sum(a11.[Sales]) AS WJXBFS1
from      [FACT_STORE_SALES1]    a11,
          [LU_STORE]           a12,
          [LU_HIST_STORE]       a13
where     a11.[STORE_SPK_ID] = a12.[STORE_SPK_ID] and
          a12.[HIST_STORE_SPK_ID] = a13.[HIST_STORE_SPK_ID]
group by  a13.[HIST_MANAGER_ID],
          a12.[HIST_STORE_SPK_ID]
  
```

The query accesses the LU_HIST_STORE table to obtain this result set.

Save the report

- Switch to Grid View.
- Save the report in the Reports folder as Current and Historical Manager information and close the report.

Create a Report on Your Own

As an extra challenge, you will create the next report on your own. The desired result set is shown at the end of the exercise.

Report Requirements

In the Reports folder, create a report that displays only historical information for managers. In other words, it should show sales for only those stores that have managers that are not the current managers. Your template should include store and manager information as well as the sales. After you run the report and view the result set, save the report as Historical Manager Information.

You need to create a filter on the MRR Flag attribute.

Report Result

Historical Manager	Historical Store	Metrics	Sales
Jim Smith	Metro West		11
Missy Wells	Metro South		15