WHITE PAPER

Modeling Best Practices

Understanding how to model to leverage Qlik Compose data warehouse automation

TABLE OF CONTENTS

EXECUTIVE SUMMARY

- Qlik's data warehouse automation provides end to end data warehouse lifecycle management including management of the data model
- The data model in Qlik Compose enables automation and near real-time data processing
- Understand best practices for common data warehouse modeling scenarios

INTRODUCTION

Qlik Compose data warehouse projects provide end to end lifecycle management of your data warehouse including capabilities to manage the central warehouse model, automate extract, transform and load (ETL) processes and restructuring data into data marts designed for bi consumption. The most important facet of any data warehouse is building a data model that supports the needs of the business and enables the conformity of data from enterprise systems.

Qlik Data Integration provides data modeling capabilities that enable real-time data warehousing and automation of the data mart design and ETL code.

This technical paper is a guide to understanding Qlik Data Integration warehouse modeling design techniques and best practices for common business scenarios.

Understanding the Qlik Compose Model

Qlik Compose model generation provides the foundational layer for your data warehouse solution. It not only serves to structure the data in an efficient, business centric fashion (leveraging data vault modeling techniques such as hubs and satellites), it also drives much of the intelligence in the ETL automation and data mart creation. Therefore, it's important to understand modelling best practices and how certain architectural decisions can impact your data warehouse projects and timelines.

Good modeling practices can:

- Reduce complexity of ETL developed by the data warehouse team
- Reduce data processing SLAs with CDC integration
- Increase the amount of data warehouse automation
- Increase the centralization data processing logic and reduce complexity of data mart design

Qlik provides four out-of-the-box methodologies for implementing the model:

1. Intelligent discovery
2. ErWin model import
3. CSV import
4. Manual-create data model with visual tools.

Note: The tips provided in this paper should be followed regardless of the chosen modeling method.

Qlik Modeling Concepts

Explaining generic database modelling concepts is out of scope for this paper, but it is important to understand that the Qlik model is defined and designed using similar modeling definitions

- **Entities**
- **Attributes**
- **Relationships**

These logical constructs are then physical implemented in the data warehouse as

- **Tables –** Entities are physically created as 1: N tables using Hubs and Satellites
- **Columns –** Attributes are physically created as columns in Hubs and Satellites
- **Relationships –** Although not physically implemented as foreign keys (FK's), relationships have a direct impact on the ETL and data mart automation functionality within Qlik.

Qlik's modeling philosophy is based on the data vault methodology. However, modeling in Qlik does not require in-depth knowledge of data vault specifics. Rather, we provide logical modeling features while automatically managing the underlying physical data vault. Note Qlik can also handle advanced data vault management scenarios such as multiple satellites to manage fast and slow-moving data. The Qlik model provides the structure for the data warehouse from which data marts are generated.

Qlik Data Marts

While modern data platforms such as Amazon Redshift, Azure Synapse and Snowflake continue to rollout more scalability features, data warehouses still need to structure their tables for efficient queries to support granular and summary analytics. Those tables are commonly referred to as data marts. Qlik provides a data mart wizard that helps you rapidly design the tables and generates all of the necessary automation code to fit your business requirements.

Qlik's notion of data marts are star-schema oriented materialized (physical) structures that are built from the central data warehouse model. The central model has a direct impact to the data mart structures that can be built and automated.

In addition, fact tables can also be built from any data warehouse table or tables related to each other in the data warehouse model. Furthermore, dimensions are selected from tables related to the data warehouse fact tables. Note that dimensions do not need to be modeled in the central data warehouse as de-normalized entities since Qlik automatically de-normalizes tables related to the dimension "root" (or grain of the dimension). Processing of this data is done in an automated, incremental fashion.

Modeling for Real-Time Data Warehousing

A common requirement for Qlik data warehouse automation solutions is to enable near real-time data warehousing through the combination of Qlik Replicate and Qlik Compose.   Data warehouses often require delivering denormalized structures in the data mart to support BI / visualization and reporting performance. This leads to complex ETL particularly related to change data capture (CDC) based processing. A best practice is to design and manage a more normalized structure for the central warehouse model while leveraging Qlik's data mart automation to handle incremental denormalization requirements. This concept is described in detail in the white paper Modeling Real-time Data Warehouses in Compose. It is recommended to review this whitepaper first as it provides a foundational concept for other modeling practices in Qlik Compose – leveraging normalization techniques to support real-time processing.

Thinking Hierarchically with Qlik Compose Models

The Qlik Compose model defines the central warehouse structures which feed data marts for data consumers. Qlik Compose provides features to design the components of a data mart (facts and dimensions) which support star and constellation designs (multiple fact tables with conformed dimensions). The design of these artifacts is derived from the central model. To enable ETL process automation, data mart design must ensure integrity in the granularity of the object (fact or dimension). For example, ensuring that automated ETL processes for a "Product" dimension does not create duplicate records when loading the dimension table. In Qlik, this is enforced by selecting an entity as the dimension root and then leveraging the relationship hierarchies to determine which entities are viable for denormalization. This means that while any entity in the model can be selected as the root (or grain) of a dimension / fact table, only entities that are *ancestors* (parents in a primary key – foreign key relationship) of the root can be automatically denormalized into the object.

Consider the Qlik Compose model below that was discovered from a source system. In the source system the *ProductDescription* table has a foreign key relationship to the *Product* table (depicted by the relationship arrowing pointing from the child to the parent).

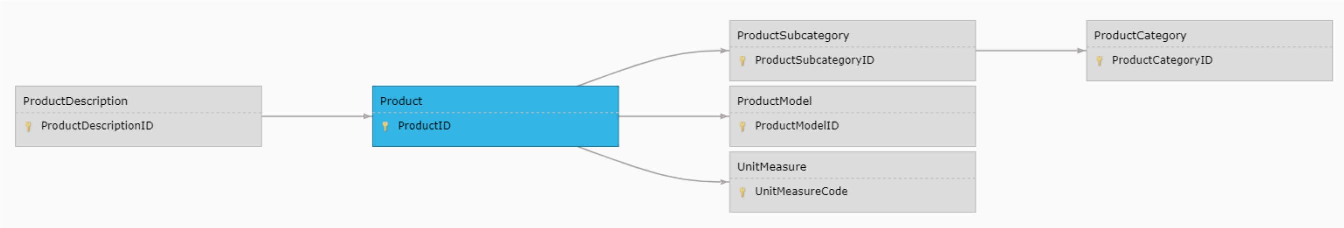


Figure 1: A normalized Qlik Compose model for product data

When defining a *Product* dimension with ETL process automation, Qlik must leverage the model and relationships to ensure there is integrity for data loaded into the *Product* dimension. Selecting the *root* entity of *Product* enables the automated denormalization of *ProductSubcategory, ProductCategory,* *ProductModel and UnitMeasure*. These are *ancestors* of the *Product* entity and thus guaranteed to not impact the granularity of a Product dimension. However, *ProductDescription* is a child of *Product* meaning it **could** have multiple records per *Product*. Therefore *ProductDescription* cannot be included in a Product dimension as its currently modeled. (Qlik Compose cannot guarantee the additional join would not duplicate entries for the Product dimension).

Resolving this (and many other modeling scenarios) requires thinking hierarchically to restructure the model to enable data mart automation. Source systems often have structures built for operational processing and flexibility which are not always appropriate for data warehousing and automation. The remainder of this paper will provide best practices to model hierarchically to resolve specific source data / business scenarios. Additional white papers are referenced that provide further details on end to end design in Qlik Compose for specific scenarios (modeling and ETL processing).

Considering Relationship Direction

A common scenario in source systems is to have one-to-one relationships between tables. Often times 1:1 cardinality define supertype-subtype relationships or even vertical partitioning of data elements (there are of course other use cases, but we will focus on those). Let's consider an example where our sales database contains orders for two types of Customers – individuals and businesses. In the source system this has been modeled leveraging supertype-subtype concepts. Discovering the source system provides the below Qlik Compose model.

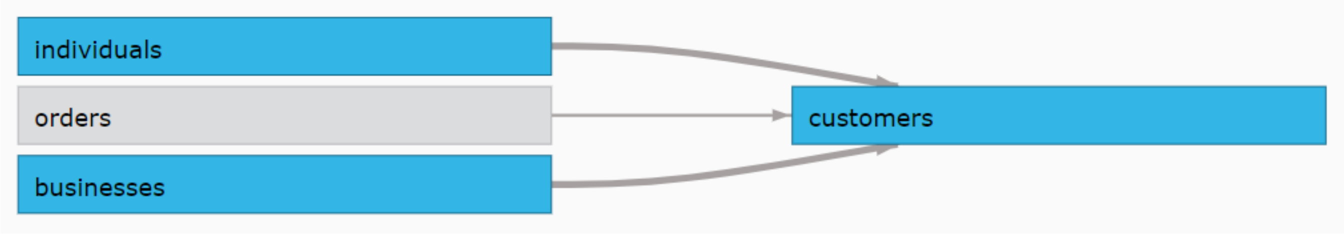


Figure 2: Supertype-subtype relationships as discovered by Qlik Compose

In this model, the *customers* entity represents the supertype (i.e. parent) and *individuals* and *businesses* are the subtypes (or children). As previously discussed when modeling in Qlik Compose you should think hierarchically. While the parent child relationships make sense in the OLTP world, for Qlik Compose, the hierarchical nature of data mart processing negates the use of the *individuals* or *businesses* entities for a *customer* dimension. Modeling these structures in Compose requires the reversal of the relationship such that customers is a child to both *individuals* and *businesses.* This is a manual process to delete the current relationships and create 2 new relationships from *customers* to *individuals* and *customers* to *businesses*. To create the new relationships navigate to the *customers* entity in Model management and select **Add Relationship** to create the foreign key relationship from *customers* to *individuals*. Select the appropriate configuration for *History type* and if required input a *Prefix* to clearly describe the relationship.

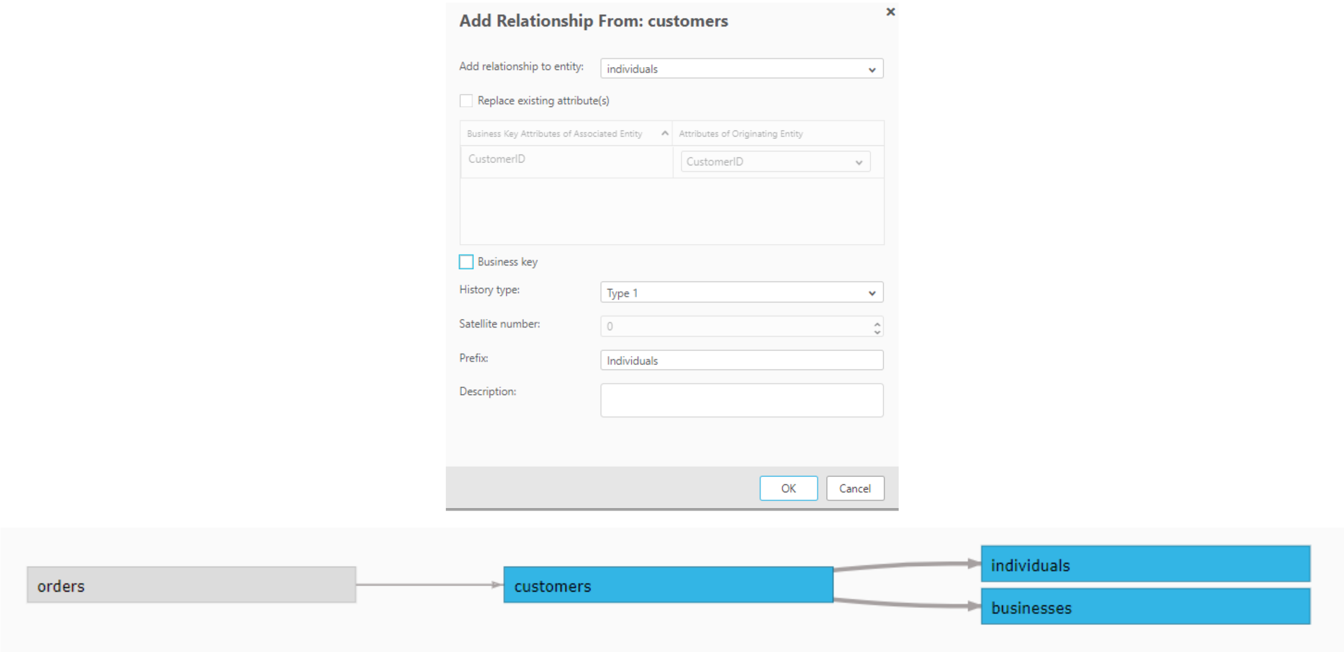


Figure 3: Relationship created from customers to individuals and the Model depicting the reversed relationships

Repeat the relationship creation to relate *customers* to *businesses.* Figure 3 shows the creation of a relationship from *customers* to *individuals* and the appropriate model in Qlik Compose. The *customers* entity can now be the root entity for a *customer* dimension and attributes from *individuals* and *businesses* can be guaranteed to not impact the integrity of the dimension grain.

The aforementioned *ProductDescription* to *Product* relationship is another example where reversing the relationship between the two entities would maintain the business model of the data while enabling data mart automation. In Figure 4, the relationship between *Product* and *ProductDescription* has been reversed. This guarantees the integrity of Product data when building the data mart objects in Compose.

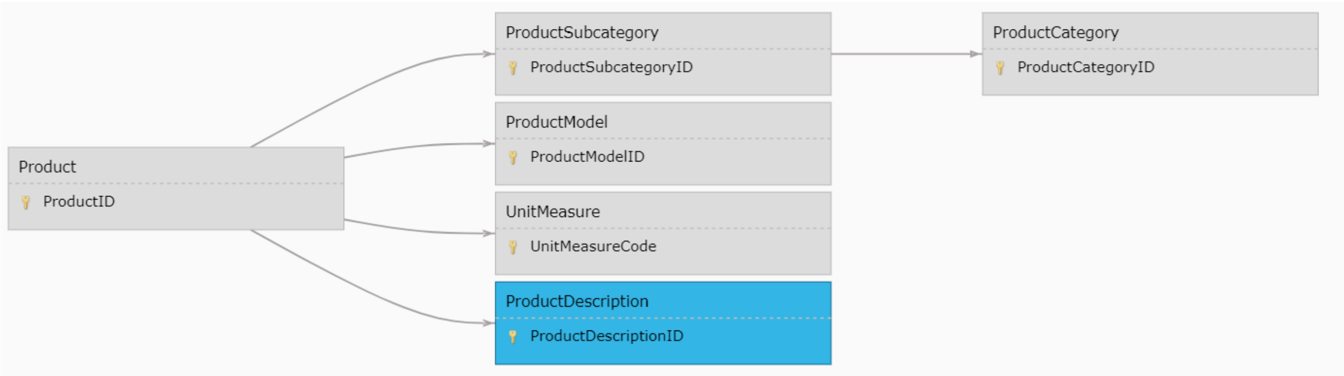


Figure 4: Model depicting the reversed relationship from ProductDescription to Product

Pivoting key-value tables

Core operational systems are often modeled in third normal form (3NF) for faster inserts, updates, and deletes. However, while this is an appropriate structure for a transactional system it does not lend itself to business intelligence (BI), dashboarding, and reporting. Although ETL tools and manual SQL scripting are potential solutions, many find them time consuming to implement, even for common data transformation tasks such as pivoting data.

Consider the simple example below. An application data source has a *CustomerContact* table that stores a distinct contact type and value for each customer. Additionally, a customer with multiple contact types are stored as multiple rows.

Key	Name	Attribute Domain/Rela...	Data Type
	CustomerID	CustomerID	Varchar(5)
	AlternateContactType	AlternateContactType	Varchar(10)
	Contact	Contact	Varchar(100)

Figure 5. CustomerContact table and model as Discovered by Compose

But what happens when our business users request reports that consist of just one row per customer as below?

ID	CustomerID	WebsiteContact	EmailContact	AltPhoneContact
1	ALFKI	www.AlfredsFutterkist...	Maria.Jones@AlfredsF...	030-007439
2	ANATR	www.AnaTrujilloEmpar...	Ms..Ana.Trujillo@AnaT...	(5) 555-471
3	ANTON	www.AntonioMorenoT...	Antonio.Moreno@Ant...	(5) 555-393
4	AROUT	www.AroundtheHorn....	Thomas.Hardy@Arou...	(171) 555-77P
5	BERGS	www.Berglundssnabb...	Christina.Berglund@B...	0921-12 34 2
6	BLAUS	www.BlauerSeeDelikat...	Hanna.Moos@BlauerS...	0621-0846
7	BLONP	www.Blondelpèreetfils...	Frédérique.Citeaux@B...	88.60.15.0
8	BOLID	www.BólidoComidasp...	Martín.Sommer@Bóli...	(91) 555 22 3

Figure 6. A Pivoted Data Set

Applying hierarchical concepts to this structure immediately tells us that the relationship between a *customer* entity and the *customer_alt_contacts* is a one-to-many relationship that precludes the contact information from being included in an automated customer dimension. The solution is to adjust the model and pivot the attributes required for analytics into a flat structure. Pivoting is the process of converting multiple rows in a single row with multiple columns and is often handled by ETL processing. However the model must also be structured appropriately. This can be done by manually creating a new entity or altering the existing entity and then "reversing the relationship" such that a new *customercontact* table is now a parent to the *customer* entity

Attributes

\+ New Attribute... Edit... Bulk Edit... Delete... Add Relationship... Show Lineage + Add to all Satellites and Hub Manage Attribut

Key	Name	Attribute Domain/Related Entity	Data Type	History	Satellite/Hub
🔑	CustomerID	CustomerID	Varchar(5)	Type 1	Hub
	WebsiteContact	Contact	Varchar(100)	Type 1	Hub
	EmailContact	Contact	Varchar(100)	Type 1	Hub
	AltPhoneContact	Contact	Varchar(100)	Type 1	Hub

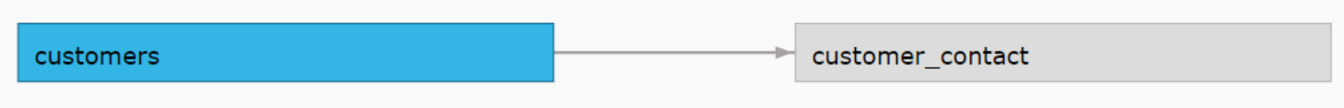


Figure 7. Restructured CustomerContact entity and reversed relationship to hold the pivoted data set

Figure 7 shows the pivoted view of data in the model. Each type of contact required in the data warehouse has been defined as an explicit column. To complete the modelling scenario we must implement a relationship reversal (as well as the entity changes) to reflect a hierarchical view of *customer* → *customer_contact* entity. This supports our business requirements to provide multiple points of contact in our data warehouse and BI solutions as well as enabling Qlik Compose's data mart automation.

This use case is defined in more detail (including ETL mapping features to support real-time data pivoting) in the ***Defining Data Models and Mappings for Pivoting Data in Qlik*** white paper on community.qlik.com.

Restructuring Associative Entities (handling many-many relationships)

Operational source systems often manage many to many business relationships via associative entities (also referred to as bridge tables). In operational systems these associative entities provide flexibility for adding additional instances of the relationships. When considering modeling these structures in a data warehouse, an associative entity could be the basis for a fact table, or it could provide for denormalization of data to represent a dimension.

In Compose any associative table that represents a basis for a fact table does not need additional design. Continuing with our sales example – an *order* can have multiple *products* and a *product* can be on multiple *orders*. As such an *order_details* entity provides the association between *orders* and *products*. Figure 8 shows the Compose model for such a scenario. There is no need to alter this model as *order_details* becomes the root for our *SalesFact* table in the data mart and allows for *orders*, *products*, and *customers* dimensions.

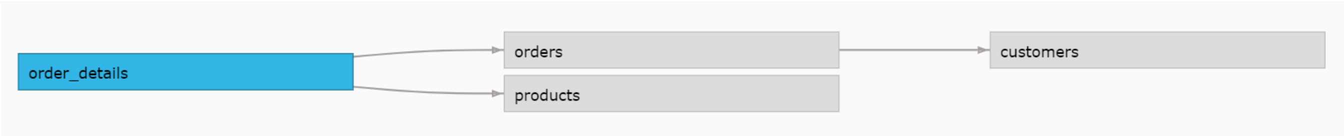


Figure 8: Associative entity the provides the basis for facts and measures

Associate entities that represent relationship instances for dimensional data however will likely need to be restructured. This is common in any data warehouse implementation and is not specific to Qlik Compose. Let's consider such a scenario, expanding on our sales data model with customers and customer addresses. Figure 9 shows the source model (discovered in Compose) for a scenario where a customer may have many addresses and each discrete address could be used by multiple customers (potentially for different reasons).

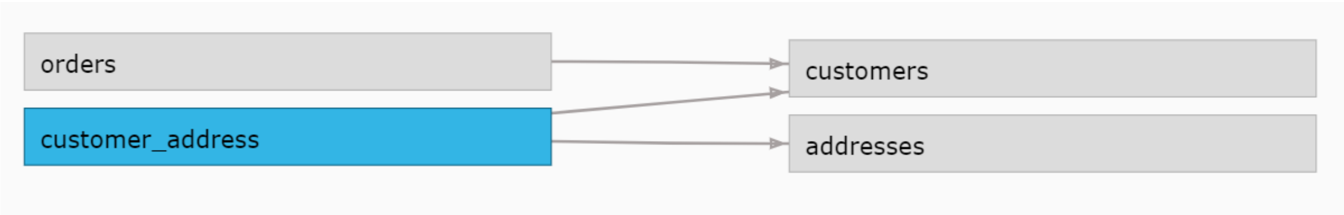


Figure 9: Associative entity handling many to many in reference data

The *customer_address* table depicts the relationship between a customer and their addresses, and how each address is used. For example customerA could use Address1 as a home address and shipping address but have Address2 as the billing address.

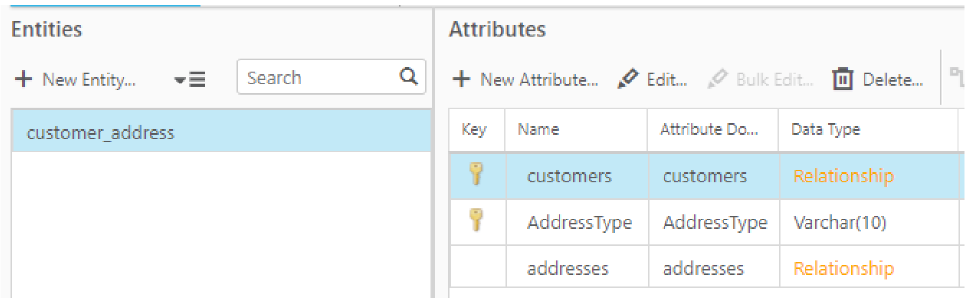


Figure 10: The customer_address table as its represented in the source

Even without Compose this type of scenario often requires de-normalizing this data to a single record or via multiple relationships so that a single customer has specific address attributes for each type of address (home, shipping, billing). In data warehousing scenarios, we understand the types of analytics and reporting required from our business data and those requirements should be used to help define the model in Compose. In our case we are concerned about reporting on home, shipping, and billing address types. Remember, for Compose modeling we must think hierarchically. This means either restructuring *customer_address* or *customers* to pivot the address usage into multiple relationships from *customer* to *address.*

Option 1 – Pivot the associative entity and create multiple relationships

One option that maintains an associative entity to manage multiple relationships is to pivot the different address types into multiple columns. This results in a customer_address entity as shown in Figure 11 where *customer_address* houses 3 role playing relationships to the *addresses* entity.

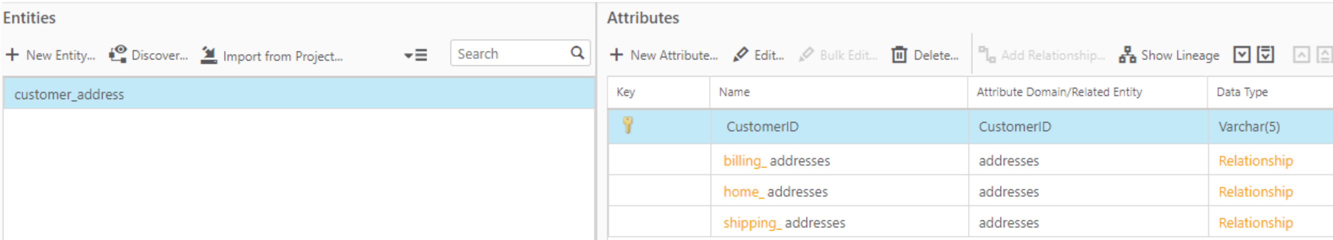


Figure 11: customer_address entity pivoted to support role-playing relationships to the address table

Figure 12 shows the hierarchical relationship structure for the altered model which now allows for the denormalization of customer and 3 addresses when designing the dimensional structures.

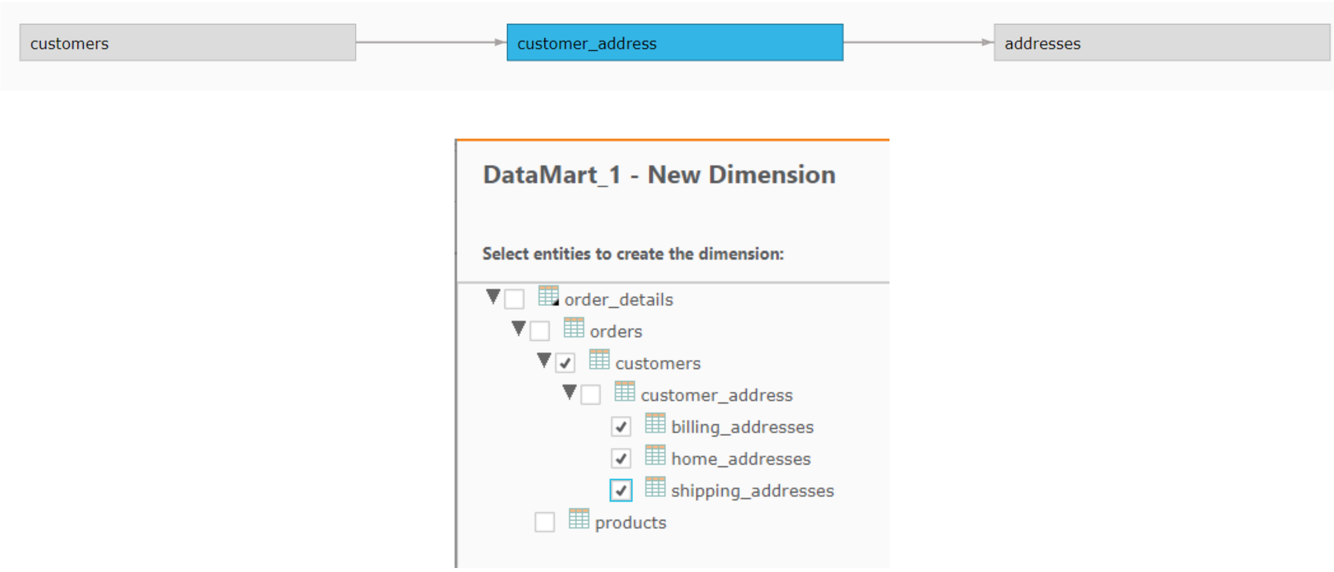


Figure 12: Hierarchical relationship from customers to address and data mart selection to automate denormalization of customers and their addresses via role-playing

Option 2 – Create multiple relationships between *Customer* and *Address*

Another option to manage the associative table in a hierarchical fashion follows the same principals as pivoting the associative entity – creation of multiple relationships. However, in this scenario those relationships will not be managed as their own entity, but rather just relationships between *customers* and *addresses*. In this scenario the *customer_address* table is deleted from the model, and 3 relationships are defined in the *customer* model to support the 3 address types that need to be managed. Figure 13 shows the adjusted customers table with the role-playing relationships and the model depiction

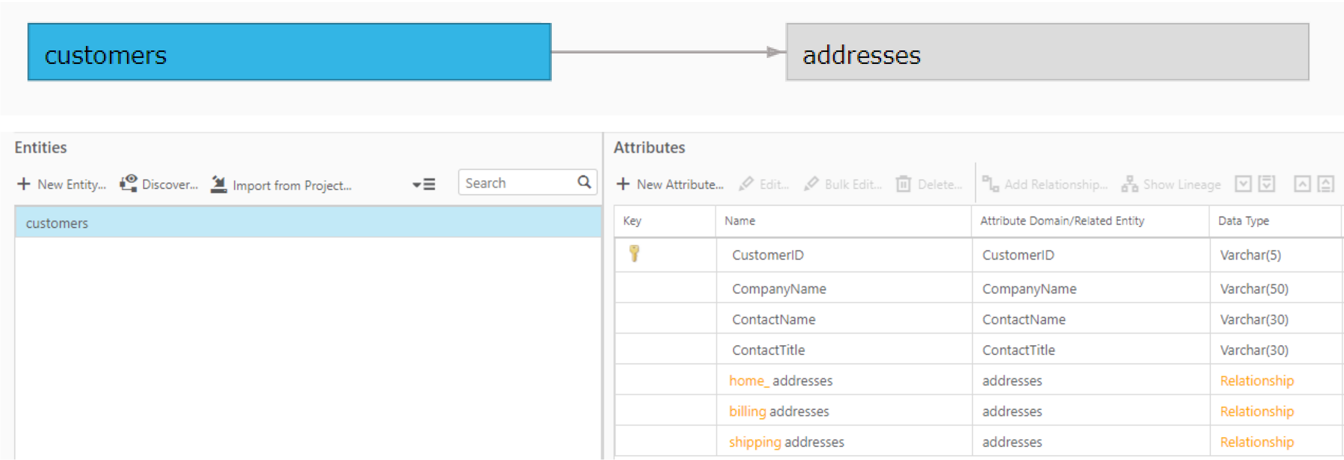


Figure 13: Customers entity with role-playing relationships and the graphic depiction of the model

This modelling method supports the hierarchical relationship requirements for Qlik Compose to automate the data mart processing and the business requirements for analytics and reporting. Note that Compose provides features to support pivoting data in near real-time through ETL mapping capabilities. This is out of scope for this white paper, but defined in further detail in the ***Defining Data Models and Mappings for Pivoting Data in Qlik*** white paper on community.qlik.com.

Managing Universal Lookup Tables and Role-Playing relationships

Operational systems such as ERP solutions often leverage universal lookup tables that provide descriptive values for multiple types of "business codes". For example instead of having discrete tables for customer type codes, customer finance categories and product type codes, they are all maintained within a single lookup table. The source application and any reports would join to the universal lookup table with filters to select the required set of records.

code_category	code_type	code_value	code_description
Customer	CustType	1	Direct Consumer
Customer	CustType	2	B2B
Product	ProdType	ASD	Services Engagement
Product	ProdType	QWE	Tangible Product
Customer	FinCat	T11	Cash/Credit Payment on Purchase
Customer	FinCat	T31	30 Day Finance Terms

Figure 14: A universal lookup table example

There are a few different methods to modeling universal lookup tables in Qlik Compose. For our example, Figure 15 represents the *all_code_lookups* table which houses different types / categories of lookup tables in a single structure.

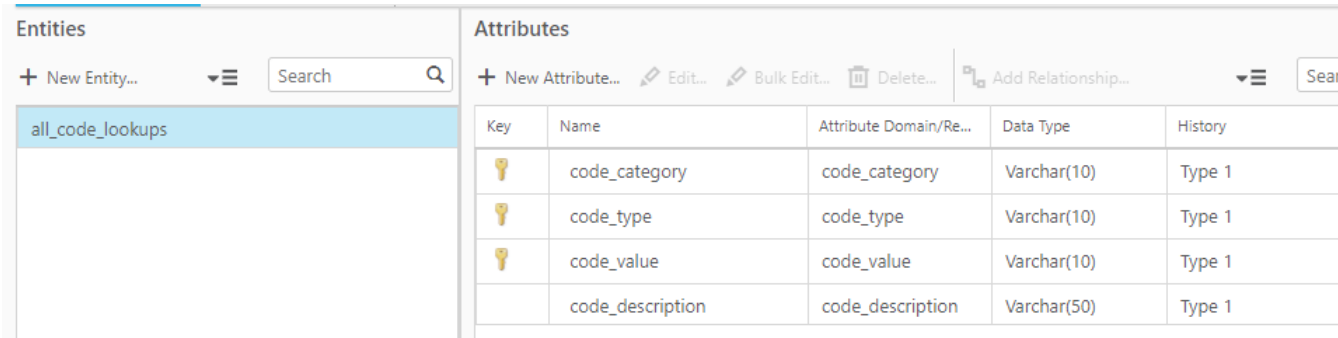


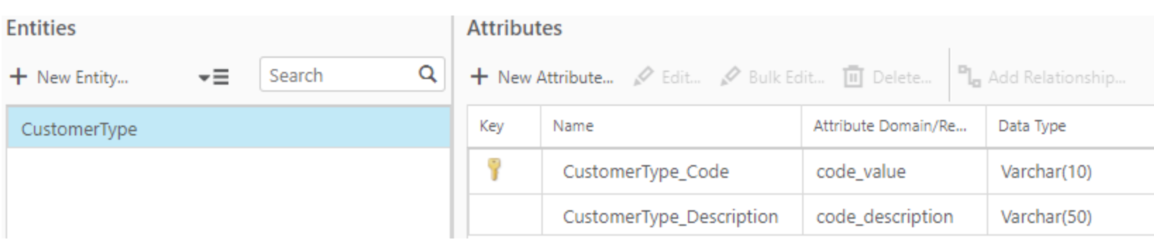
Figure 15: Universal Lookup table as discovered from the source

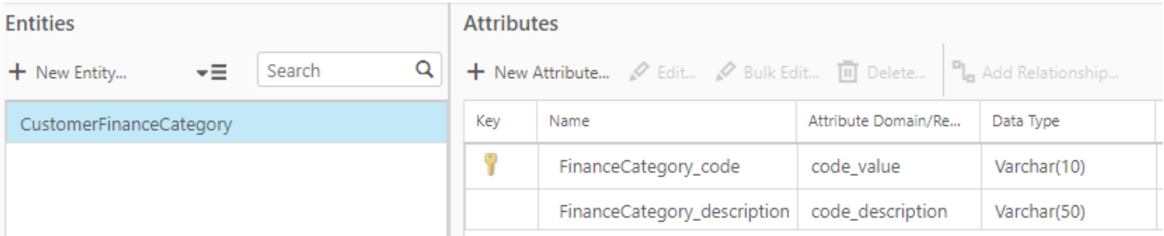
This table relates to *customer* and *product* entities with descriptive values for customer type and finance category and product type codes (figure 16). The code_category and code_type columns provide the context for the lookup code and descriptive values.

Figure 16: The Compose model maintaining the universal lookup

This entity could be left as is with multiple relationships defined from the customer / product entities to the *all_code_lookups* table. However, it is often better in a data warehouse scenario to model the individual lookup tables as part of the core model so that the model is more descriptive.

The below images show individual lookup tables modeled for *CustomerType* and *CustomerFinanceCategory* to house the code and descriptive values for each lookup domain.





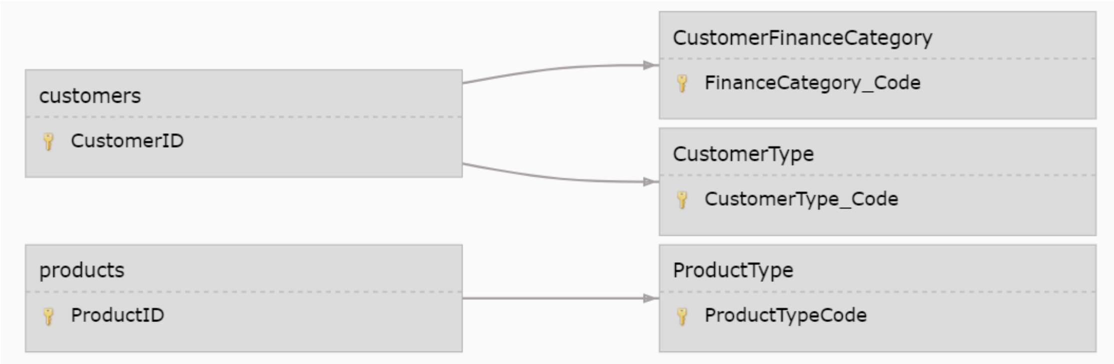


Figure 17: Discrete entities defined for individual lookup domains and the final model

The final model is depicted in Figure 17 to show the 3 modeled lookup tables and their relationships to the *customers* and *products* entities.

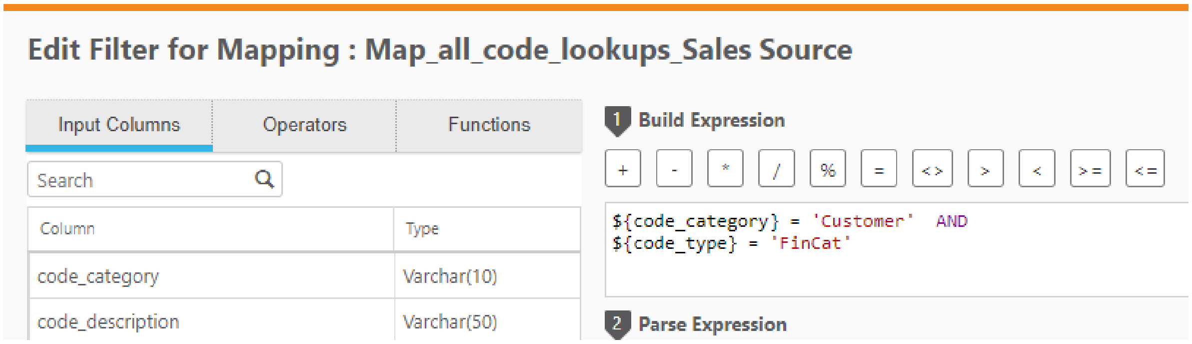


Figure 18: Mapping filters applied for CustomerFinanceCategory mapping

The ETL processing for these is simple. Each mapping will source from the *all_code_lookups* table delivered by Replicate with simple filters for the *code_category* and *code_value* to reduce the data for the specific lookup domain (figure 18). This technique provides a more descriptive model definition for lookup tables instead of having a generic lookup table that is used for multiple purposes.

Modeling Hierarchies

Hierarchies play an important part in analytic solutions and data warehouse design. They often provide aggregation and drill paths for data. There are 3 common types of hierarchies that need to be managed in a data warehouse solution: Standard (or fixed-level) hierarchies, Self-referencing (or recursive) hierarchies and Ragged Hierarchies.

Standard or Fixed Level Hierarchy in Compose

Standard or fixed level hierarchies such as the *Product→ ProductSubcategory→ ProductCategory* shown in the Compose model below are simple to design and model. These could be modeled in a single entity with attributes for each level and multiple ETL mappings, each loading a specific level of the hierarchy, or modelled in a normalized fashion as shown below. (This use case is covered in the modeling for real-time data warehouse white paper referenced earlier).

Figure 19: Fixed hierarchy modeled as multiple entities in Compose

Fixed-level hierarchies can be modeled in a normalized fashion in the Compose model which supports flexible design patterns in the data mart layer.

3 Level Self-Referencing (Recursive) Hierarchy

Self-referencing relationships are recursive hierarchies where a single entity is typically involved with a parent-child relationship managed within said entity. An example of this is shown in Figure 20 with an *employee* entity. Each *employee* has a manager who is also an employee. Thus the self-referencing relationship.

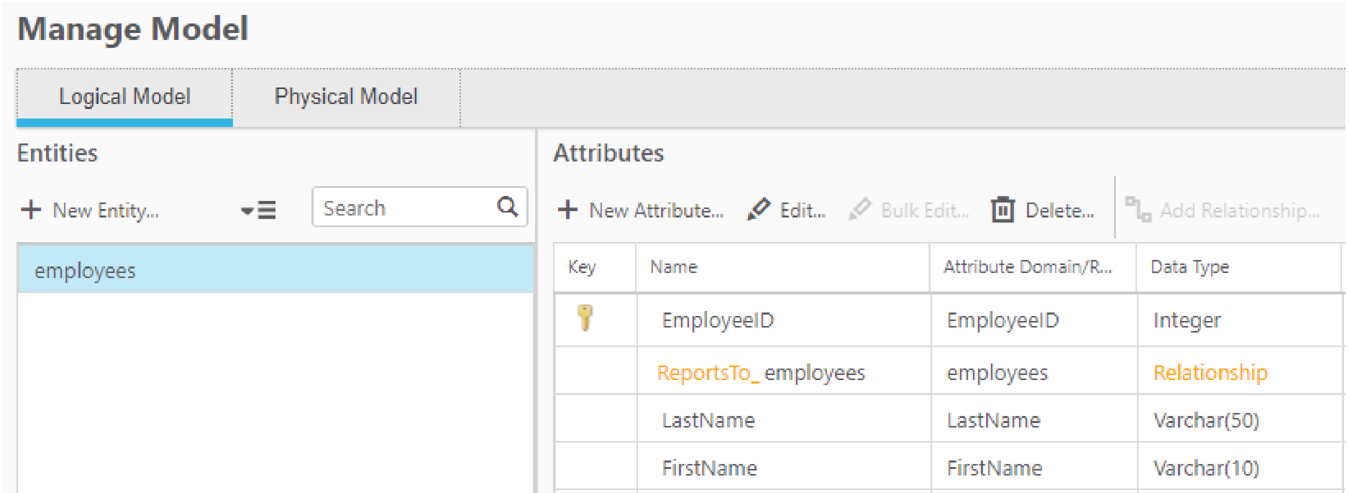


Figure 20: Self-referencing relationship for single level hierarchy

If your dimensional requirement is to support up to three levels of your hierarchy (e.g. employee > Supervisor > Manager) and is a self-referencing relationship, this can be handled automatically by Compose in both the warehouse model and the data mart level. Simply create a self-referencing relationship (as shown in the *employees* entity in Figure 20). Compose will automatically enable two levels of recursion when deploying the data mart.

Then when creating the dimension the additional levels will be available for denormalization.

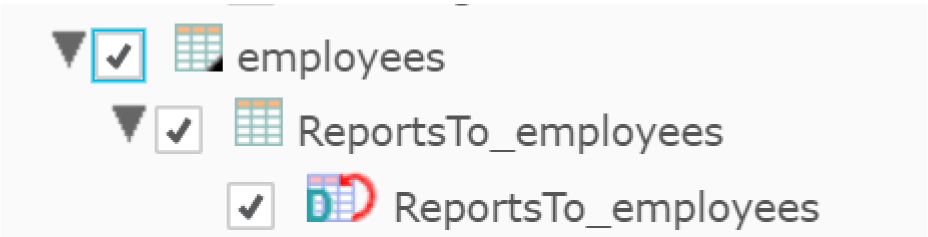


Figure 21: Creating a dimension with a self-referencing relationship enables two level recursion / denormalization

Ragged Hierarchies

Modelling and ETL changes come into play when the relationship is recursive (beyond three levels) or we are managing a ragged hierarchy. A ragged (or jagged) hierarchy is a hierarchy with an un-even number of levels for each branch of the hierarchy. Because many BI solutions cannot handle ragged hierarchies when creating dashboards and analytics we may need to standardize the hierarchy levels [in the data mart dimension object]. Let us consider the simple organizational hierarchy depicted below.

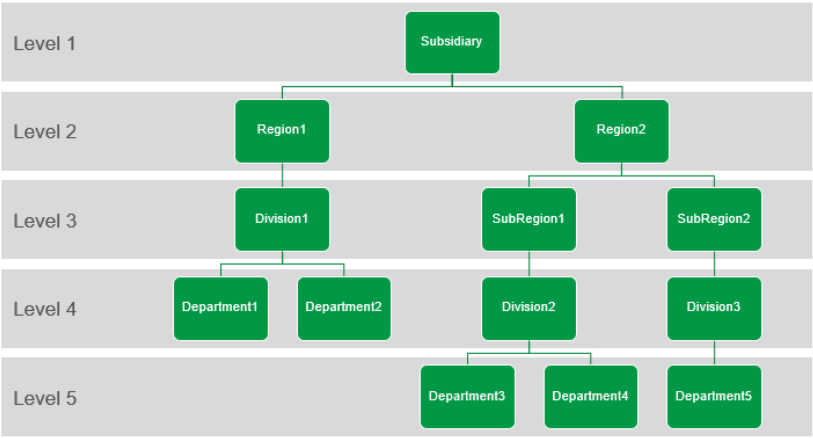


Figure 22: Ragged hierarchy example

While each branch of the hierarchy has *Departments, Divisions and Regions* only Region2 has *SubRegions*. This is considered a ragged hierarchy because the hierarchy depth is different (or un-even) for the two regions. In our analytics, any aggregated metrics on a specific "level" in the hierarchy would be inconsistent. You can see that below Level2 (*Regions)* we would be mixing different organization levels (for example divisions and subregions in *Level 3.*)

Modelling a ragged hierarchy therefore requires determining specific levels for the hierarchy and standardizing on those levels. Figure 23 shows how the ragged hierarchy has been standardized to support 5 levels with an explicit name for each level.

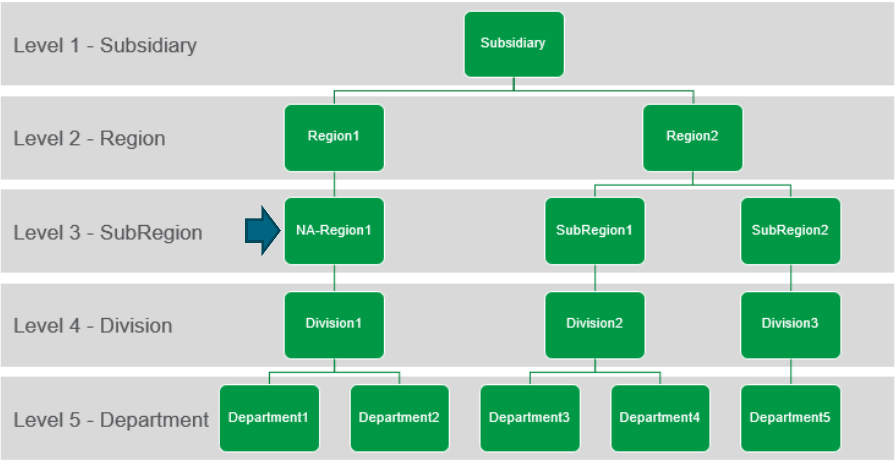


Figure 23: Ragged hierarchy forced into a fixed level hierarchy

The data must be processed to accommodate the standard structure. Note the ***NA-Region1*** , which does not exist in the source and has been created as part of the ETL. Since *Region1* does not have a SubRegion, a "dummy entry" has been created. While the label for this value in the hierarchy could be different, the concept of rolling the ***Region*** down and creating a “subregion” is common in ensuring standard hierarchies. This does 2 things for us. It ensures that when BI solutions are looking at aggregates of sales or other key metrics across the organization data is aggregated at the same “business” grain. It also provides a placeholder and a label to let data consumers know that a subregion does not exist for Region1.

Now that we have conceptually resolved the ragged nature of the hierarchy the model can be designed in one of two ways. Each *level* in the hierarchy can be an entity in the model with appropriate relationships or a single entity can be defined with appropriate attributes for each level of the hierarchy. Designing the model as a normalized set of entities (figure 24) where each level is its own entity provides more flexibility in defining dimensions as we can have a single organizational dimension, or a dimension defined at a specific grain of the hierarchy (for example a Region dimension).

Figure 24: Compose Model with each hierarchy level as an entity

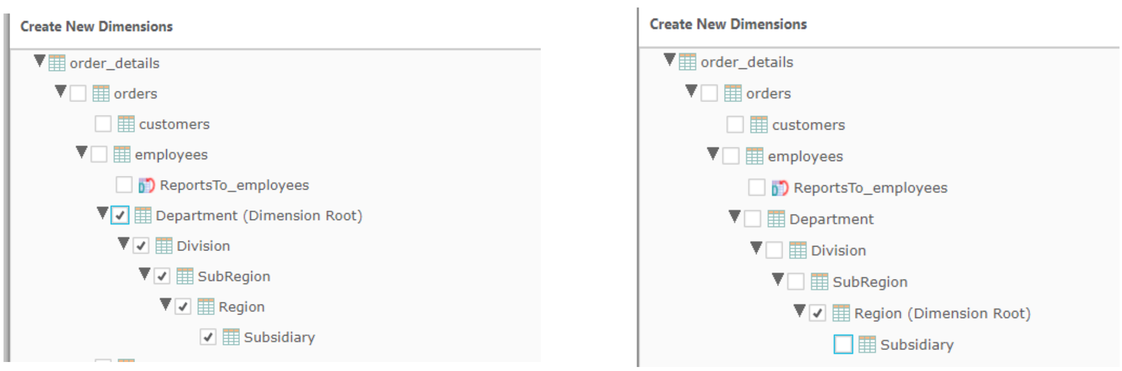


Figure 25: Creating a single Organizational Dimension versus a Dimension for a specific level (Region)

Figure 25 shows the creating a single de-normalized dimension or selecting a specific level of the hierarchy for more granular dimensional control. Since organizational hierarchies do not change frequently these are typically (and recommended to be) loaded in batch oriented processes only (not using CDC) and can leverage query based sources if required to fill in the missing levels. Standardizing the hierarchy levels simplifies the design and ETL process and provides better anchor points for analysis and aggregation for consumers.

Merging Multiple Source Systems

Data warehouses consolidate and harmonize data from multiple source systems. The following will discuss modeling practices in Compose when using the model discovery feature. These practices can be applied to any model implementation however, if you have modelled your data warehouse “manually” (whether outside of Compose in a data modeling tool or in Compose) then your data model is likely a canonical model built to support your business and merging multiple source systems is a matter of defining the logic in the ETL mappings.

Data harmonization scenarios can vary, this paper will cover four common requirements for conforming data from multiple source systems.

1. Union scenario - the same data domain is managed by multiple source systems with no overlap in the rows. In this scenario there are often common attributes and source specific attributes that should be modeled.

2. Join scenario – the same data domain is managed by multiple source systems with discrete attributes being managed by each source and a common key between the sources.

3. Mastering scenario – the same data domain is managed by multiple source systems with overlap in both attributes and records. The warehouse is required to make decisions on which attributes to use from specific sources.

4. Discrete domains scenario – data domains are managed by independent sources systems. There is no overlap within the domain, but common business keys to join / reference across domains.

The last scenario - discrete data domains managed by different source systems is easy to model. For example sales data from an online portal and product data from a product data management solution. Modeling this scenario is as simple as discovering the source tables and manually creating the relationships between product and sales entities on the common natural keys. Using model Discovery to implement the other scenarios often requires additional customizations to the data model.

Modeling the Union Scenario

A union scenario is when a single data domain comes from multiple sources with no overlap in the data. There are often a set of attributes that are common between sources and a set that are unique to each source system. This is common for sources of transactional or fact data. In our scenario sales data comes from multiple systems that have different structures. There is no relationship between the data – each record from both sources represents a distinct sale of a product. When this occurs we must create standardized entities to manage sales data and not simply Discover each source. The source system models are shown below.

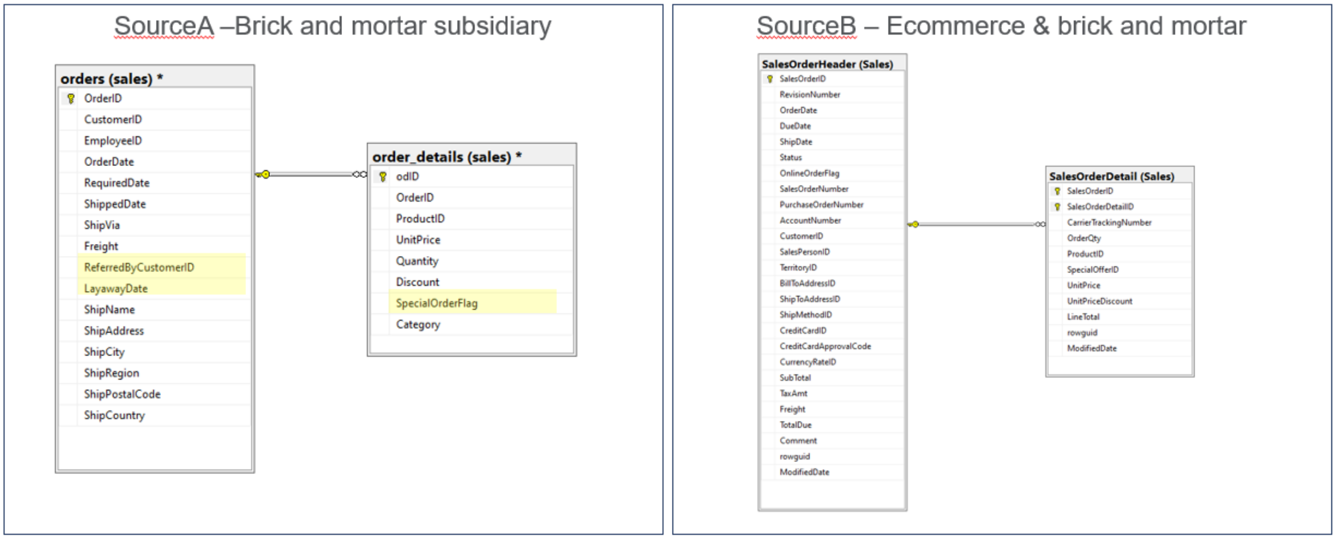


Figure 26: Creating a single Organizational Dimension versus a Dimension for a specific level (Region)

While the structures are similar in this case, there are common attributes (albeit named differently such as *Quantity* and *OrderQty*, *RequiredDate* and *DueDate*) and there are source specific attributes / business concepts that do not overlap (such as *TaxAmt, TerritoryID etc.)*. In this case *SourceA* represents data from our brick-and-mortar store system and *SourceB* from a subsidiary of the business that sells through online channels as well as direct to consumers in stores (hence the *OnlineOrderFlag* attribute).

When modeling this scenario in Compose (as with any data warehouse implementation), we should design a canonical model for sales data. Creating a "sales header" and "sales detail" entity that houses the common and uncommon elements between the two sources. We do not want to create these entities for each discrete source. Since SourceB has the super-set of attributes we will discover that source and use it as the starting point for the model. If you review the header tables you can see that most of the columns for SourceA map to columns in SourceB. This mapping will be performed in the ETL mappings (as it would in most data warehouse deployments). The changes we will make to the model will be to

a) ensure the uniqueness of keys from each source

b) remove attributes that are not needed in the data warehouse (e.g. ModifiedDate)

c) add attributes that exist in SourceA, but not SourceB

d) adjust data types and attribute names if required

Ensuring uniqueness in the natural keys for both the header and details is as simple as adding a SourceSystem attribute to the key. This ensures that sales id 1 is unique for SourceA and Source.

Business requirements dictate which attributes should be removed and added. In our case the additional columns have been highlighted in the model – *LayawayDate* and *ReferredByCustomerID* for sales header data and *SpecialOrderFlag* for sales details. These attributes can be manually added to the respective entities in the model (see figure 27 for the SalesOrderHeader design).

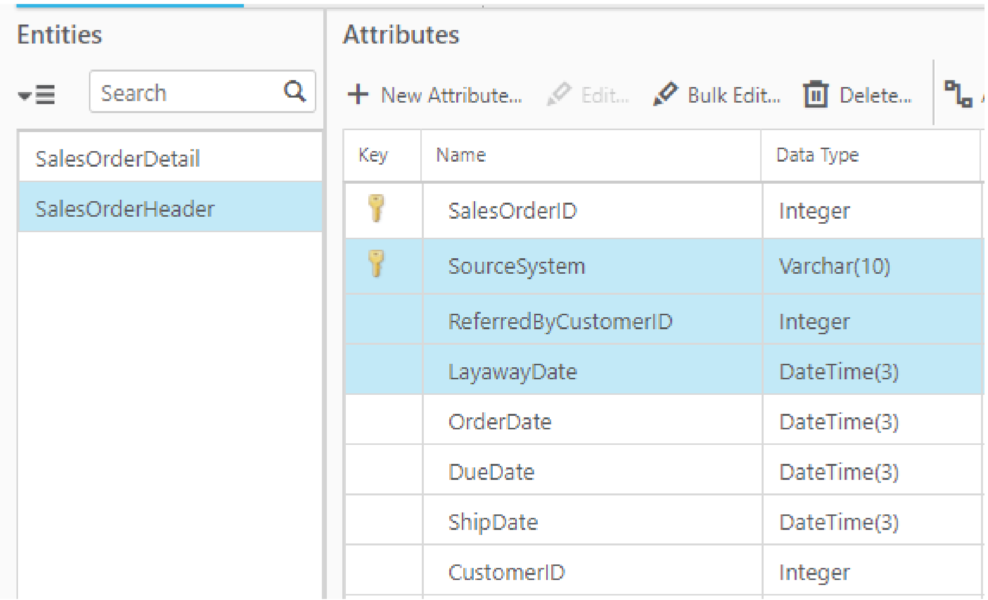


Figure 27: SalesOrderHeader has had the highlighted attributes added to fit the business requirements

Once added, the model for sales data (accommodating overlapping and source specific attributes) is complete.

Merging the data is now a function of the ETL in Compose.   The ETL mappings will need to be created for SourceA and adjusted for SourceB to correctly map attributes, conform data values and provide default values where required. For example special orders are only accommodated by our SourceA subsidiary.   Data mapped from SourceB will have a *SpecialOrderFlag* value of “N” for all records.

While not required for our example the final step would be to adjust any data types and attribute names. Data types should be altered to ensure they fit the data from either source and attribute names altered to ensure they represent common business vernacular.

Modeling the JOIN scenario

The simplest way to implement (and best practice for near real-time data warehousing) is to Discover the tables from both sources and define relationships between the discovered entities on common natural keys (remember to *Think Hierarchically* in the creation of your relationships). This retains a normalized structure for real-time processing (refer to the Modeling for Real-Time Data Warehousing section of this white paper).

For example – given a *Customers* entity from one source system and demographic data from our marketing department we can simply create a relationship in the model and allow Compose to handle either de-normalizing the structures into a single dimension or managing each as independent dimensions based on consumer requirements. Figure 28 shows the discovered entities.

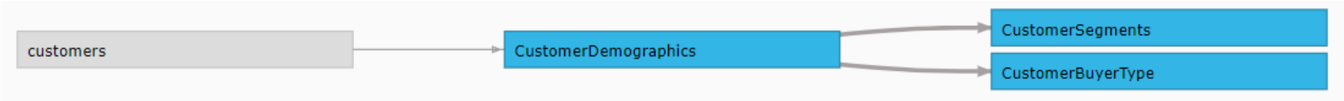


Figure 28: customers from SourceA and customer demographic tables (highlighted) from the marketing department

The relationship between *customers* and *CustomerDemographics* was manually added to support joining these entities into a single *Customers* dimension (if required).

This method adds entities to the main warehouse model but enables the most automation in model design and ETL mapping generation. Another method to implement would be to model the *customers* entity with all the required attributes. Demographic attributes would need to be manually added to the discovered *customers* entity (figure 28).

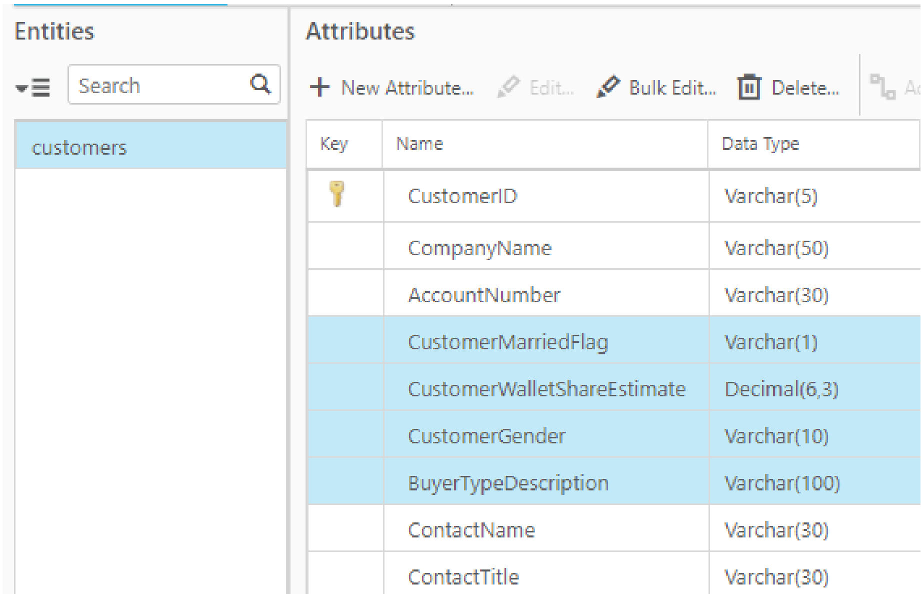


Figure 29: customers entity with demographic attributes (highlighted) added

With the model defined, the ETL mappings would manage the transformation requirements to load the data warehouse. Qlik Compose provides multiple features to manage these joins (source queries, lookups, or partial mappings). These concepts are not covered in this white paper.

Modeling the Mastering scenario

Qlik Compose is **not** a master data management solution. However, data warehouses often have to manage data that overlaps from multiple source system and implement simple, rule based decisions on which source attribute should be used for analytics. These rules are typically as simple as "if SourceA has a value for attribute use it, else use SourceB's value". Often times these are soft business rules which may change over time. For this scenario, the vault concepts used by Compose for the warehouse layer can provide flexibility to managing the attributes and changing soft business rules. We will cover two solutions with an example scenario where customer data is provided by multiple source systems.

Review the source table structures shown in figure 30.

customers (sourceA)

Column Name	Condensed Type	Nullable
CustomerID	nvarchar(5)	No
CompanyName	nvarchar(50)	Yes
ContactName	nvarchar(30)	Yes
ContactTitle	nvarchar(30)	Yes
Address	nvarchar(60)	Yes
City	nvarchar(15)	Yes
Region	nvarchar(15)	Yes
PostalCode	nvarchar(10)	Yes
Country	nvarchar(15)	Yes
Phone	nvarchar(24)	Yes
Fax	nvarchar(24)	Yes
DELETED_FLAG	varchar(50)	Yes

customer (sourceB)

Column Name	Condensed Type	Nullable
CustomerKey	int	No
PersonID	int	Yes
StoreID	int	Yes
TerritoryID	int	Yes
AccountNumb...	nvarchar(10)	No
rowguid	nvarchar(36)	No
ModifiedDate	datetime2(3)	No
AddressId	int	Yes
CustomerTypeId	int	Yes
FirstName	nvarchar(100)	Yes
LastName	nvarchar(100)	Yes
BusinessName	nvarchar(100)	Yes
last_update_dtm	datetime2(6)	Yes
CustomerID	varchar(10)	Yes

Figure 30: Customer entities in two source systems

In this case there is overlap of customer information between the two systems. E.g. *John Doe*'s customer information is available in both sourceA and sourceB while *Jane Doe* only exists in sourceA. When conforming data in this fashion we have to ensure we have a common value that can be used as the unique or natural key for the model that can cross-reference the two sources.

In this case the primary key for each system is obviously different with sourceA having nvarchar(5) and sourceB having integer columns. Notice that sourceB also has a *CustomerID* field with a varchar datatype. This represents the *CustomerID* primary key field in sourceA and is the value that enables cross-referencing the two systems. This may not always be available, and you may have a scenario where a cross-reference table manages the relationship between the two sources. Either way, the ability to link records from both systems is a requirement. *(Note that this ability to link records is often the purpose of a master data management or data quality solution in the context of the data warehouse. This is out of scope for core Qlik Compose functionality however Qlik Compose could orchestrate these external processes or additional custom code as part of the end to end data processing requirements for cross-reference creation and data harmonization).*

Since there is overlap in the data, but not all customers from sourceA exist in sourceB this is not a case of discovering the entities and applying relationships. Instead, we will discover sourceA and create duplicate attributes for source. Recall that Compose converts the logical model into HUB and SATELLITE tables to manage business keys and type 2 attributes. When duplicating logical attributes for each source we will also designate that the data is managed in different SATELLITES so that the type 2 nature of data is managed by each source system. For this example we will focus a few customer attributes – *CompanyName* and *ContactName*.

Key	Name	Data Type	Attribute Domain/Related Entity	History	Satellite/Hub
🔑	CustomerID	Varchar(5)	CustomerID	Type 1	Hub
	sourceA_ CompanyName	Varchar(50)	CompanyName	Type 2	1
	sourceA_ ContactName	Varchar(30)	ContactName	Type 2	1
	sourceB_ CompanyName	Varchar(50)	CompanyName	Type 2	2
	sourceB_ ContactName	Varchar(30)	ContactName	Type 2	2

Figure 31: Attributes defined for each source system, managed in different Satellites

Figure 31 shows the *customers* entity. Attributes *CompanyName* (BusinessName column in source) and *ContactName* (a concatenation of FirstName and LastName from source) have been created for each source. Note the satellite is different for each. This design will result in the physical tables shown in figure 32. Since this example only focuses on a few columns it may not be necessary to implement multiple satellites. However in real-world scenarios you would be managing more attributes than this and separating into multiple satellites reduces storage requirements as changes are managed across the sources.

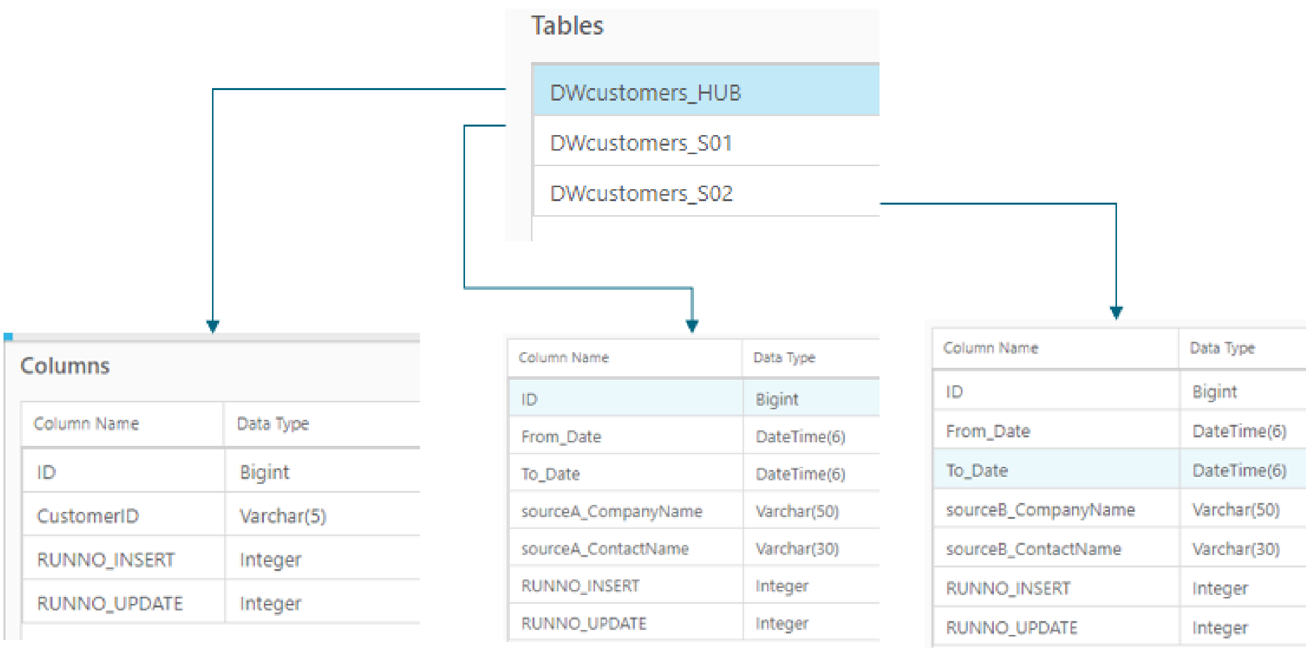


Figure 32: Physical tables (1 HUB and 2 satellites) are defined by Compose from the logical model

The model now manages each source within a single logical entity. The soft business rules (which can change over time) can now be managed in the data mart layer in Compose using the expression builder. For example the below shows the expression to leverage the CompanyName from sourceB if it exists, but the ContactName from sourceA using a simple isnull expression.

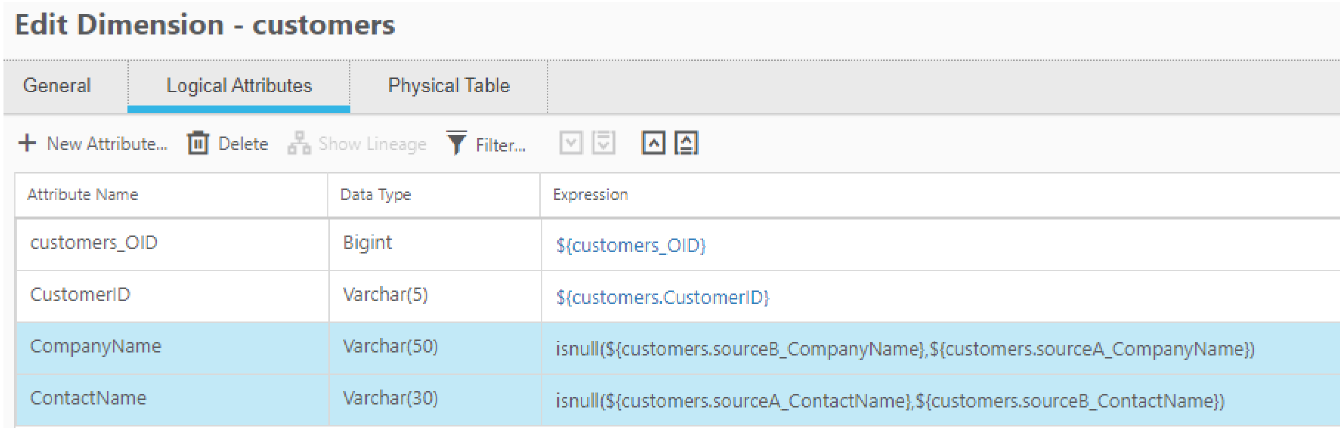


Figure 33: Dimension applying soft business rules

This solution requires a common identifier between the two customer source tables. As mentioned earlier, it is common for a cross-reference to be used to manage these relationships if the data is not readily available in the source. When using a cross-reference table the lookup can be performed within the ETL to resolve the CustomerID and CustomerKey to a common identifier or it can be modeled.

Figure 34: A cross-reference table managing relationships between the customer tables

Figure 34 shows the model using a cross-reference entity (*CustomerXRef*). The *orders* entity has been left in the image to depict how other tables should be related to the customer tables. All the relationships should be assigned to the *CustomerXRef* entity as that allows the customer dimension to be built using the *CustomerXRef* as the root of the dimension. (Enforcing dimension granularity and again, *Thinking Hierarchically*). Defining and loading this entity is commonly performed by retrieving data from a master data management or data quality solution that performs fuzzy matching and merging or by a query based Compose mapping when fuzzy matching is not required.

While we have covered four of the common scenarios in modeling data from multiple source systems, there are other scenarios when merging data from multiple systems. In these instances the general idea is to model the logical entities to fit your business and leverage ETL features in Compose to handle the processing of the data

Reserved Words to Avoid in the Model

Qlik Compose converts your logical model to physical structures using components of the data vault pattern. Compose also added specific columns for surrogate keys and for operational metadata. Compose has a set of reserved words that should not be used for attributes these are –

- **ID –** the surrogate column managed by Compose.
- **RUNNO_INSERT –** the ETL run number that inserted the record
- **RUNNO_UPDATE –** the ETL run number that updated the record
- **FD –** the *from date* for Type 2 records. The name of this column can be altered in Project settings based on your requirements (e.g. **FROM_DATE, EFF_DT, EFFECTIVE_DATE** etc.). The name you enter in the project settings will become "reserved".
- **TD –** the *to date* for Type 2 records. The name of this column can be altered in project settings based on your requirements (e.g. **TO_DATE, END_DATE** etc.). The name you enter in the project settings will become "reserved".

Additional Content

This white paper focuses on specific modeling scenarios with Qlik Compose data warehouse projects and some general best practices to enable automation throughout the warehouse design and ETL automation process. There is additional content for Qlik Compose covering different aspects of the solution from deeper dives into specific modeling and ETL use cases to handling deletes and much more. These are available on the Qlik Community Qlik for Data Warehouses pages that can be found in the Documents and Video section.

Conclusion

Qlik's data warehouse automation provides capabilities to manage the data warehouse lifecycle. Modeling of the data warehouse is a critical step in understanding your business requirements and enabling ETL and data mart automation. This paper has covered some general best practices and some specific scenarios (merging data from multiple sources) to leverage the most out of Qlik Composes automation features for managing the data warehouse and the data mart. The general principles of defining the data model are to model the entities to support your business and data warehouse requirements while following the principles detailed herein. The approaches described in this paper explain how to think hierarchically when building the data model in Compose and how to handle common modeling scenarios as you migrate from an operational to analytics model.

About Qlik

Qlik's vision is a data-literate world, one where everyone can use data to improve decision-making and solve their most challenging problems. Only Qlik offers end-to-end, real-time data integration and analytics solutions that help organizations access and transform all their data into value. Qlik helps companies lead with data to see more deeply into customer behavior, reinvent business processes, discover new revenue streams, and balance risk and reward. Qlik does business in more than 100 countries and serves over 50,000 customers around the world. **qlik.com**

LEAD WITH DATA