

Understand star schema and the importance for Power BI

This article targets Power BI Desktop data modelers. It describes star schema design and its relevance to developing Power BI data models optimized for performance and usability.

This article isn't intended to provide a complete discussion on star schema design. For more details, refer directly to published content, like **The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling** (3rd edition, 2013) by Ralph Kimball et al.

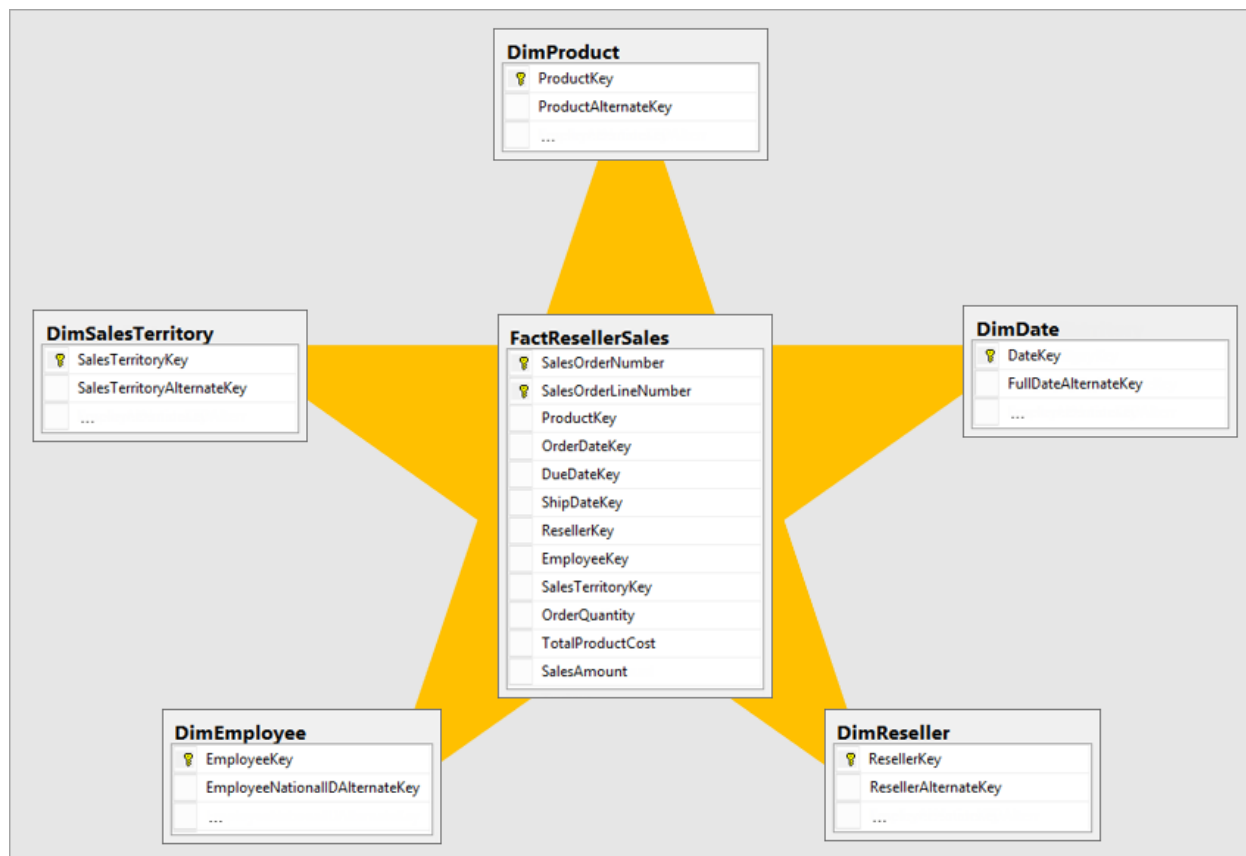
Star schema overview

Star schema is a mature modeling approach widely adopted by relational data warehouses. It requires modelers to classify their model tables as either *dimension* or *fact*.

Dimension tables describe business entities—the *things* you model. Entities can include products, people, places, and concepts including time itself. The most consistent table you'll find in a star schema is a date dimension table. A dimension table contains a key column (or columns) that acts as a unique identifier, and descriptive columns.

Fact tables store observations or events, and can be sales orders, stock balances, exchange rates, temperatures, etc. A fact table contains dimension key columns that relate to dimension tables, and numeric measure columns. The dimension key columns determine the *dimensionality* of a fact table, while the dimension key values determine the *granularity* of a fact table. For example, consider a fact table designed to store sale targets that has two dimension key columns **Date** and **ProductKey**. It's easy to understand that the table has two dimensions. The granularity, however, can't be determined without considering the dimension key values. In this example, consider that the values stored in the **Date** column are the first day of each month. In this case, the granularity is at month-product level.

Generally, dimension tables contain a relatively small number of rows. Fact tables, on the other hand, can contain a very large number of rows and continue to grow over time.



Star schema relevance to Power BI models

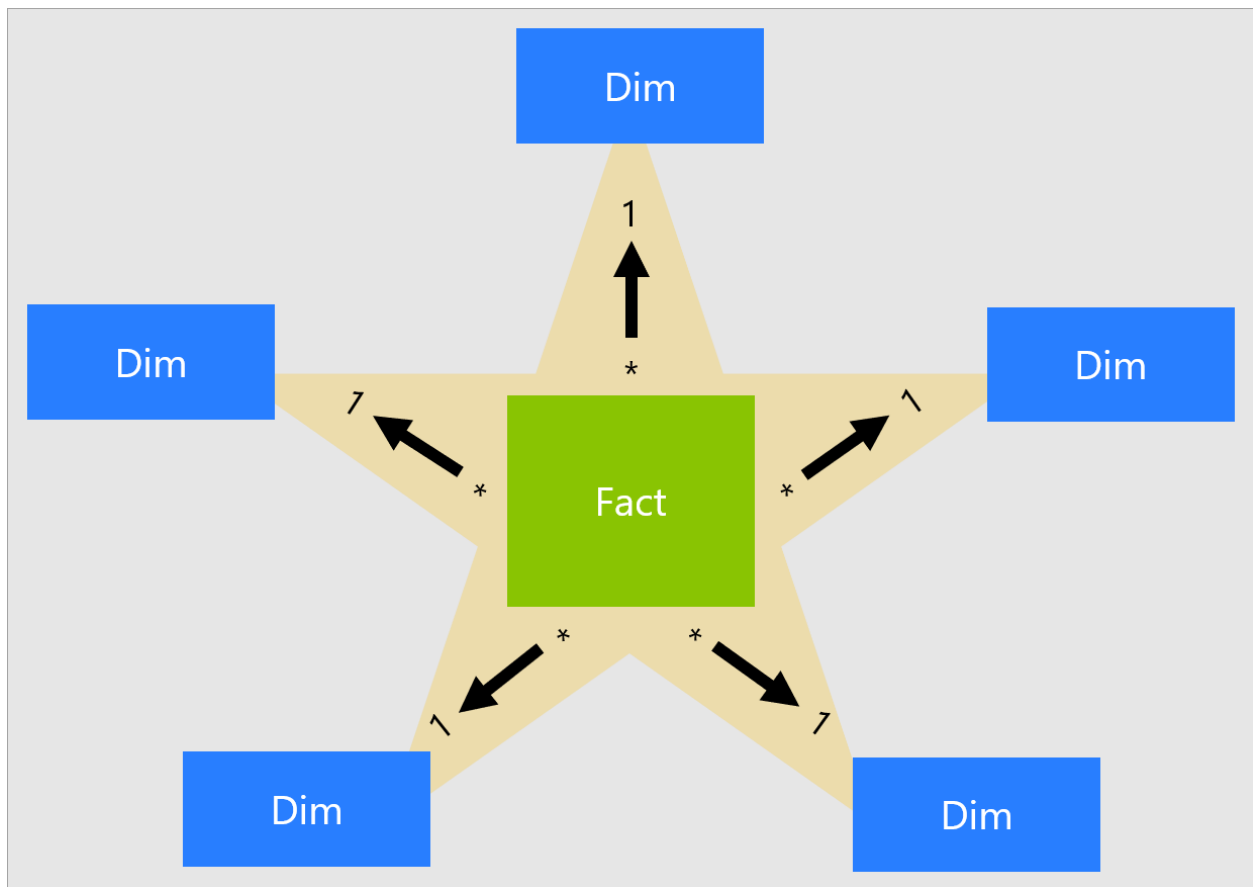
Star schema design and many related concepts introduced in this article are highly relevant to developing Power BI models that are optimized for performance and usability.

Consider that each Power BI report visual generates a query that is sent to the Power BI model (which the Power BI service calls a dataset). These queries are used to filter, group, and summarize model data. A well-designed model, then, is one that provides tables for filtering and grouping, and tables for summarizing. This design fits well with star schema principles:

- Dimension tables support *filtering* and *grouping*
- Fact tables support *summarization*

There's no table property that modelers set to configure the table type as dimension or fact. It's in fact determined by the model relationships. A model relationship establishes a filter propagation path between two tables, and it's the **Cardinality** property of the relationship that determines the table type. A common relationship cardinality is *one-to-*

many or its inverse *many-to-one*. The "one" side is always a dimension-type table while the "many" side is always a fact-type table. For more information about relationships, see [Model relationships in Power BI Desktop](#).



A well-structured model design should include tables that are either dimension-type tables or fact-type tables. Avoid mixing the two types together for a single table. We also recommend that you should strive to deliver the right number of tables with the right relationships in place. It's also important that fact-type tables always load data at a consistent grain.

Lastly, it's important to understand that optimal model design is part science and part art. Sometimes you can break with good guidance when it makes sense to do so.

There are many additional concepts related to star schema design that can be applied to a Power BI model. These concepts include:

- [Measures](#)
- [Surrogate keys](#)
- [Snowflake dimensions](#)

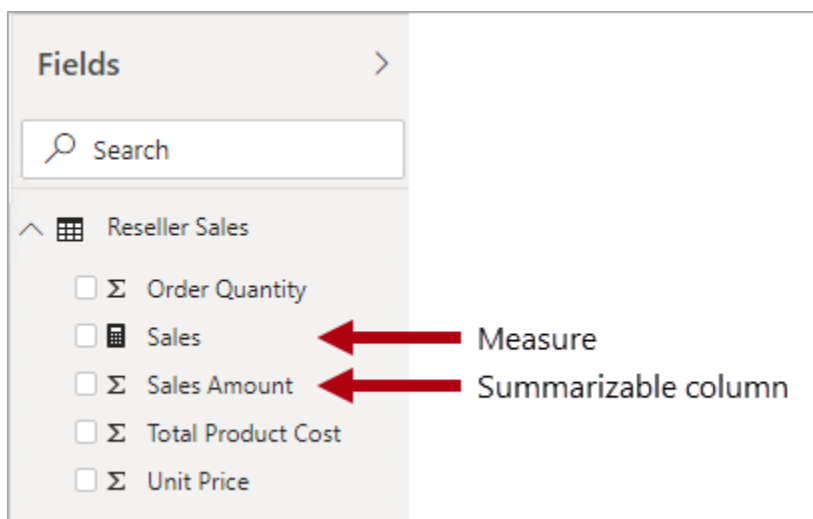
- [Role-playing dimensions](#)
- [Slowly changing dimensions](#)
- [Junk dimensions](#)
- [Degenerate dimensions](#)
- [Factless fact tables](#)

Measures

In star schema design, a **measure** is a fact table column that stores values to be summarized.

In a Power BI model, a **measure** has a different—but similar—definition. It's a formula written in [Data Analysis Expressions \(DAX\)](#) that achieves summarization. Measure expressions often leverage DAX aggregation functions like SUM, MIN, MAX, AVERAGE, etc. to produce a scalar value result at query time (values are never stored in the model). Measure expression can range from simple column aggregations to more sophisticated formulas that override filter context and/or relationship propagation. For more information, read the [DAX Basics in Power BI Desktop](#) article.

It's important to understand that Power BI models support a second method for achieving summarization. Any column—and typically numeric columns—can be summarized by a report visual or Q&A. These columns are referred to as *implicit measures*. They offer a convenience for you as a model developer, as in many instances you do not need to create measures. For example, the Adventure Works reseller sales **Sales Amount** column could be summarized in numerous ways (sum, count, average, median, min, max, etc.), without the need to create a measure for each possible aggregation type.



However, there are three compelling reasons for you to create measures, even for simple column-level summarizations:

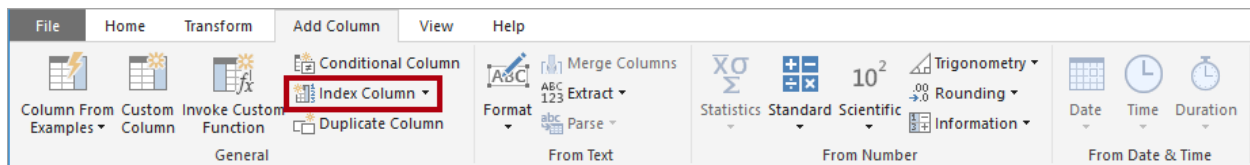
- When you know your report authors will query the model by using [Multidimensional Expressions \(MDX\)](#), the model must include *explicit measures*. Explicit measures are defined by using DAX. This design approach is highly relevant when a Power BI dataset is queried by using MDX, because MDX can't achieve summarization of column values. Notably, MDX will be used when performing [Analyze in Excel](#), because PivotTables issue MDX queries.
- When you know your report authors will create Power BI paginated reports using the MDX query designer, the model must include explicit measures. Only the MDX query designer supports [server aggregates](#). So, if report authors need to have measures evaluated by Power BI (instead of by the paginated report engine), they must use the MDX query designer.
- When you need to ensure that your report authors can only summarize columns in specific ways. For example, the reseller sales **Unit Price** column (which represents a per unit rate) can be summarized, but only by using specific aggregation functions. It should never be summed, but it's appropriate to summarize by using other aggregation functions like min, max, average, etc. In this instance, the modeler can hide the **Unit Price** column, and create measures for all appropriate aggregation functions.

This design approach works well for reports authored in the Power BI service and for Q&A. However, Power BI Desktop live connections allow report authors to show hidden fields in the **Fields** pane, which can result in circumventing this design approach.

Surrogate keys

A **surrogate key** is a unique identifier that you add to a table to support star schema modeling. By definition, it's not defined or stored in the source data. Commonly, surrogate keys are added to relational data warehouse dimension tables to provide a unique identifier for each dimension table row.

Power BI model relationships are based on a single unique column in one table, which propagates filters to a single column in a different table. When a dimension-type table in your model doesn't include a single unique column, you must add a unique identifier to become the "one" side of a relationship. In Power BI Desktop, you can easily achieve this requirement by creating a [Power Query index column](#).

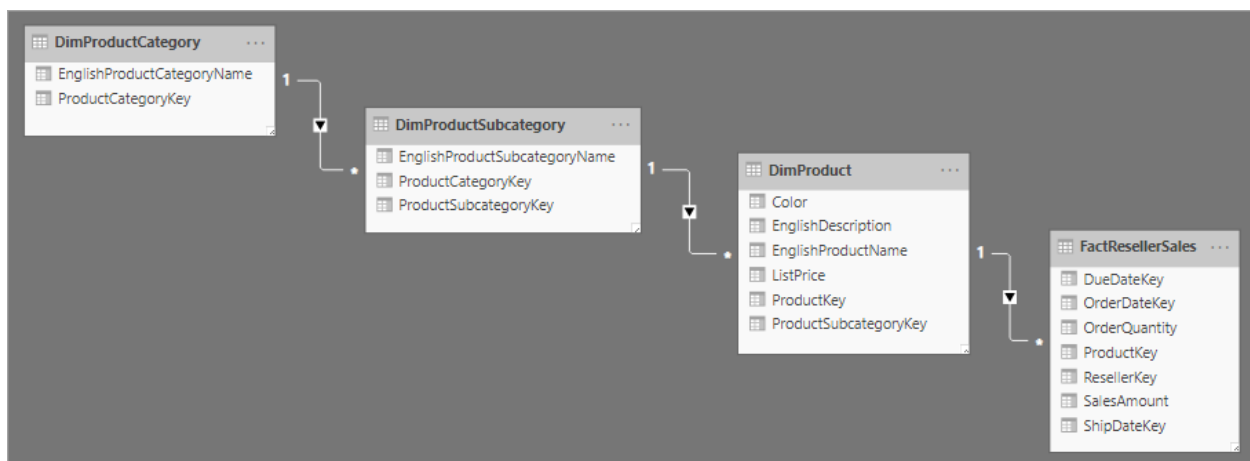


You must merge this query with the "many"-side query so that you can add the index column to it also. When you load these queries to the model, you can then create a one-to-many relationship between the model tables.

Snowflake dimensions

A **snowflake dimension** is a set of normalized tables for a single business entity. For example, Adventure Works classifies products by category and subcategory. Categories are assigned to subcategories, and products are in turn assigned to subcategories. In the Adventure Works relational data warehouse, the product dimension is normalized and stored in three related tables: **DimProductCategory**, **DimProductSubcategory**, and **DimProduct**.

If you use your imagination, you can picture the normalized tables positioned outwards from the fact table, forming a snowflake design.

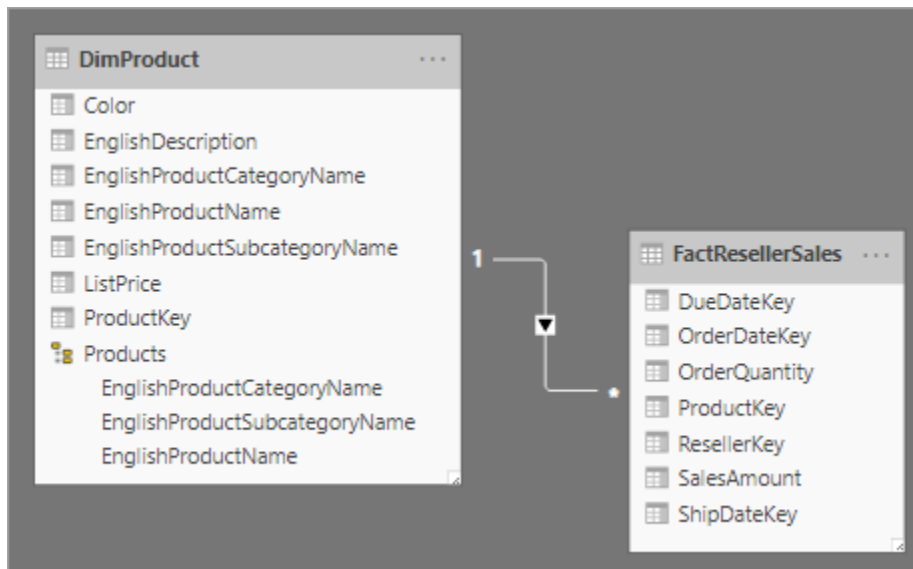


In Power BI Desktop, you can choose to mimic a snowflake dimension design (perhaps because your source data does) or integrate (denormalize) the source tables into a single model table. Generally, the benefits of a single model table outweigh the benefits of multiple model tables. The most optimal decision can depend on the volumes of data and the usability requirements for the model.

When you choose to mimic a snowflake dimension design:

- Power BI loads more tables, which is less efficient from storage and performance perspectives. These tables must include columns to support model relationships, and it can result in a larger model size.
- Longer relationship filter propagation chains will need to be traversed, which will likely be less efficient than filters applied to a single table.
- The **Fields** pane presents more model tables to report authors, which can result in a less intuitive experience, especially when snowflake dimension tables contain just one or two columns.
- It's not possible to create a hierarchy that spans the tables.

When you choose to integrate into a single model table, you can also define a hierarchy that encompasses the highest and lowest grain of the dimension. Possibly, the storage of redundant denormalized data can result in increased model storage size, particularly for very large dimension tables.



Slowly changing dimensions

A **slowly changing dimension** (SCD) is one that appropriately manages change of dimension members over time. It applies when business entity values change over time, and in an ad hoc manner. A good example of a *slowly* changing dimension is a customer dimension, specifically its contact detail columns like email address and phone number. In contrast, some dimensions are considered to be *rapidly* changing when a dimension attribute changes often, like a stock's market price. The common design approach in these instances is to store rapidly changing attribute values in a fact table measure.

Star schema design theory refers to two common SCD types: Type 1 and Type 2. A dimension-type table could be Type 1 or Type 2, or support both types simultaneously for different columns.

Type 1 SCD

A **Type 1 SCD** always reflects the latest values, and when changes in source data are detected, the dimension table data is overwritten. This design approach is common for columns that store supplementary values, like the email address or phone number of a customer. When a customer email address or phone number changes, the dimension table updates the customer row with the new values. It's as if the customer always had this contact information.

A non-incremental refresh of a Power BI model dimension-type table achieves the result of a Type 1 SCD. It refreshes the table data to ensure the latest values are loaded.

Type 2 SCD

A **Type 2 SCD** supports versioning of dimension members. If the source system doesn't store versions, then it's usually the data warehouse load process that detects changes, and appropriately manages the change in a dimension table. In this case, the dimension table must use a surrogate key to provide a unique reference to a *version* of the dimension member. It also includes columns that define the date range validity of the version (for example, **StartDate** and **EndDate**) and possibly a flag column (for example, **IsCurrent**) to easily filter by current dimension members.

For example, Adventure Works assigns salespeople to a sales region. When a salesperson relocates region, a new version of the salesperson must be created to ensure that historical facts remain associated with the former region. To support accurate historic analysis of sales by salesperson, the dimension table must store versions of salespeople and their associated region(s). The table should also include start and end date values to define the time validity. Current versions may define an empty end date (or 12/31/9999), which indicates that the row is the current version. The table must also define a surrogate key because the business key (in this instance, employee ID) won't be unique.

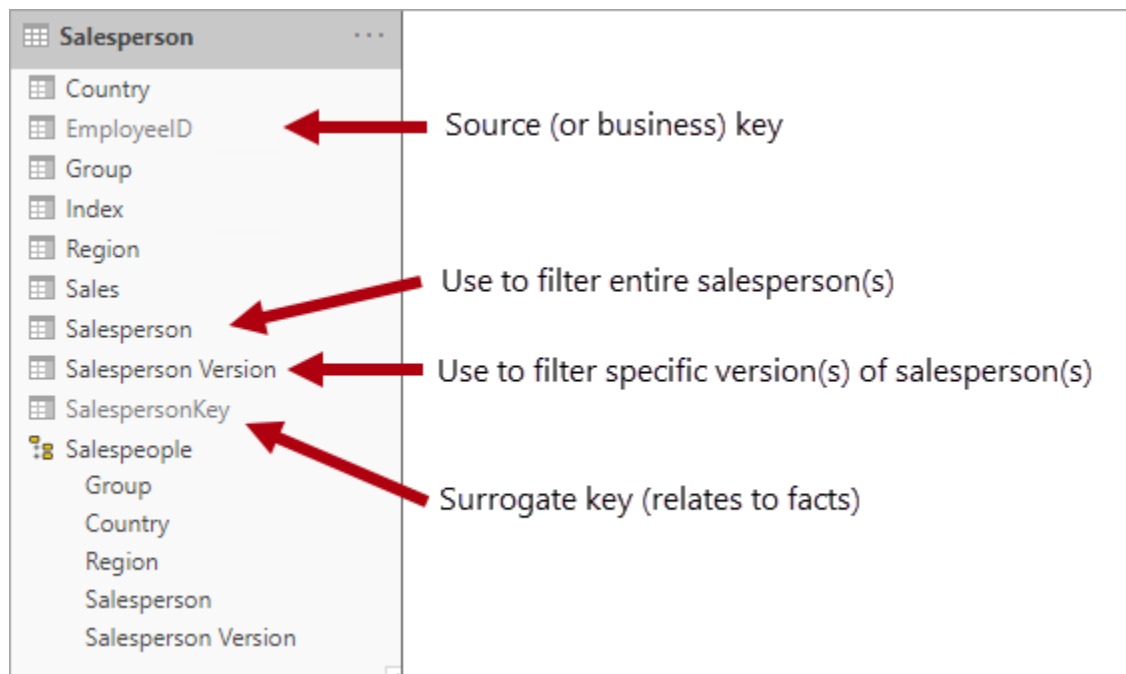
It's important to understand that when the source data doesn't store versions, you must use an intermediate system (like a data warehouse) to detect and store changes. The table load process must preserve existing data and detect changes. When a change is detected, the table load process must expire the current version. It records these changes by updating the **EndDate** value and inserting a new version with

the **StartDate** value commencing from the previous **EndDate** value. Also, related facts must use a time-based lookup to retrieve the dimension key value relevant to the fact date. A Power BI model using Power Query can't produce this result. It can, however, load data from a pre-loaded SCD Type 2 dimension table.

The Power BI model should support querying historical data for a member, regardless of change, and for a version of the member, which represents a particular state of the member in time. In the context of Adventure Works, this design enables you to query the salesperson regardless of assigned sales region, or for a particular version of the salesperson.

To achieve this requirement, the Power BI model dimension-type table must include a column for filtering the salesperson, and a different column for filtering a specific version of the salesperson. It's important that the version column provides a non-ambiguous description, like "Michael Blythe (12/15/2008-06/26/2019)" or "Michael Blythe (current)". It's also important to educate report authors and consumers about the basics of SCD Type 2, and how to achieve appropriate report designs by applying correct filters.

It's also a good design practice to include a hierarchy that allows visuals to drill down to the version level.



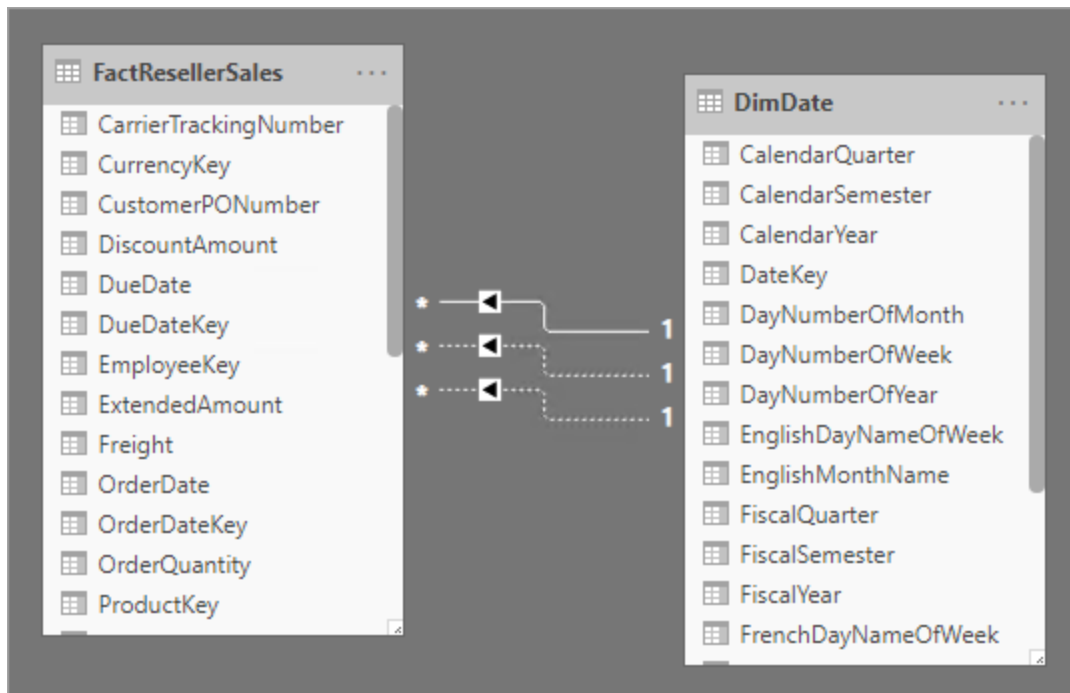
Group	Country	Region	Salesperson	Salesperson Version	Sales
North America	United States	Northeast	David Campbell	David Campbell (12/15/2008-06/26/2019)	\$52,307.11
				David Campbell (06/27/2019-Current)	\$119,807.56
				Total	\$172,114.67
			Pamela Ansman-Wolfe	Pamela Ansman-Wolfe (03/03/2007-Current)	\$222,087.01
				Total	\$222,087.01
			Tete Mensa-Annan	Tete Mensa-Annan (01/09/2005-Current)	\$23,098.4
				Total	\$23,098.4
			Total		\$417,300.08
		Total			\$417,300.08
Total					\$417,300.08

Role-playing dimensions

A **role-playing dimension** is a dimension that can filter related facts differently. For example, at Adventure Works, the date dimension table has three relationships to the reseller sales facts. The same dimension table can be used to filter the facts by order date, ship date, or delivery date.

In a data warehouse, the accepted design approach is to define a single date dimension table. At query time, the "role" of the date dimension is established by which fact column you use to join the tables. For example, when you analyze sales by order date, the table join relates to the reseller sales order date column.

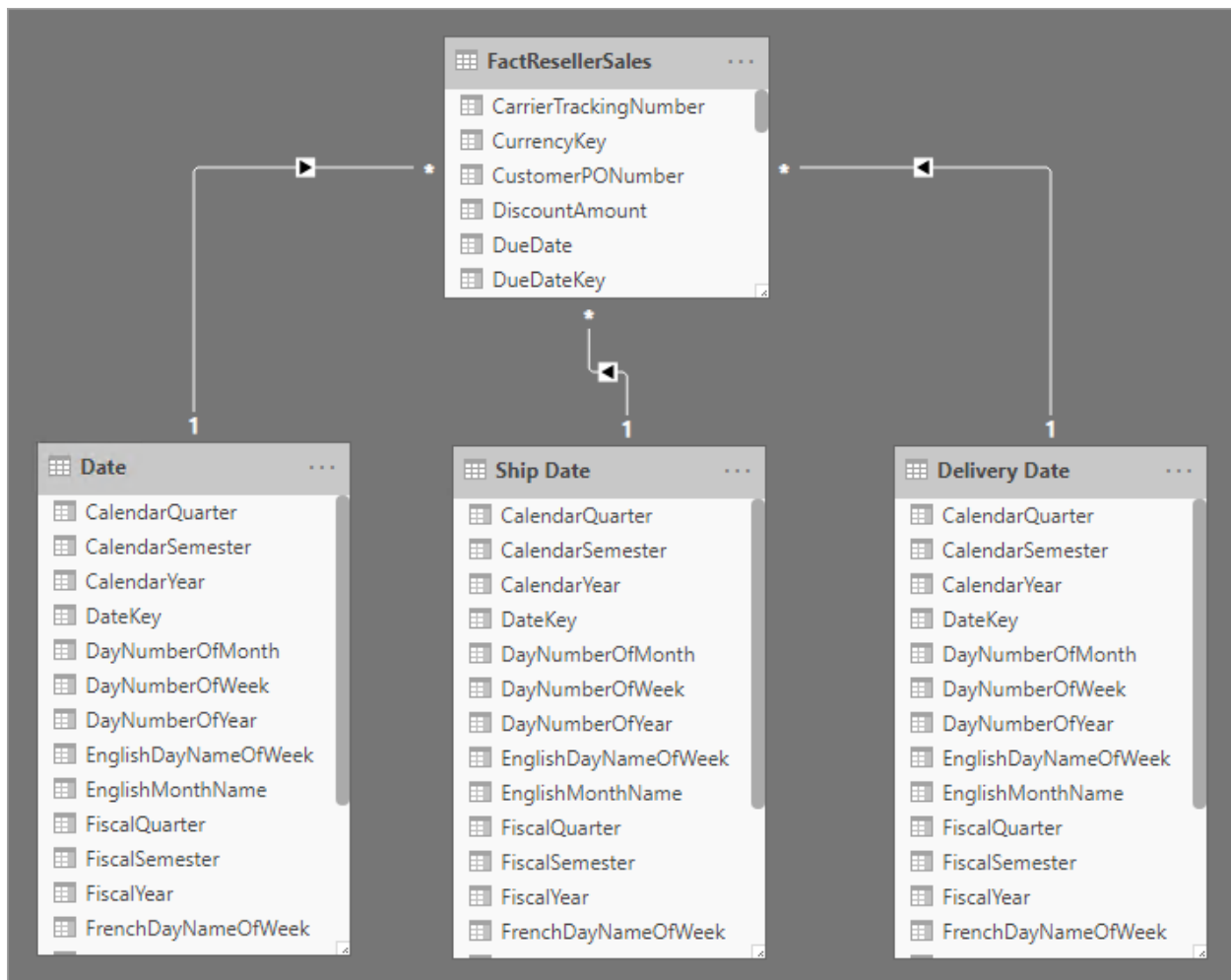
In a Power BI model, this design can be imitated by creating multiple relationships between two tables. In the Adventure Works example, the date and reseller sales tables would have three relationships. While this design is possible, it's important to understand that there can only be one active relationship between two Power BI model tables. All remaining relationships must be set to inactive. Having a single active relationship means there is a default filter propagation from date to reseller sales. In this instance, the active relationship is set to the most common filter that is used by reports, which at Adventure Works is the order date relationship.



The only way to use an inactive relationship is to define a DAX expression that uses the [USERELATIONSHIP function](#). In our example, the model developer must create measures to enable analysis of reseller sales by ship date and delivery date. This work can be tedious, especially when the reseller table defines many measures. It also creates **Fields** pane clutter, with an overabundance of measures. There are other limitations, too:

- When report authors rely on summarizing columns, rather than defining measures, they can't achieve summarization for the inactive relationships without writing a report-level measure. Report-level measures can only be defined when authoring reports in Power BI Desktop.
- With only one active relationship path between date and reseller sales, it's not possible to simultaneously filter reseller sales by different types of dates. For example, you can't produce a visual that plots order date sales by shipped sales.

To overcome these limitations, a common Power BI modeling technique is to create a dimension-type table for each role-playing instance. You typically create the additional dimension tables as [calculated tables](#), using DAX. Using calculated tables, the model can contain a **Date** table, a **Ship Date** table and a **Delivery Date** table, each with a single and active relationship to their respective reseller sales table columns.



This design approach doesn't require you to define multiple measures for different date roles, and it allows simultaneous filtering by different date roles. A minor price to pay, however, with this design approach is that there will be duplication of the date dimension table resulting in an increased model storage size. As dimension-type tables typically store fewer rows relative to fact-type tables, it is rarely a concern.

Observe the following good design practices when you create model dimension-type tables for each role:

- Ensure that the column names are self-describing. While it's possible to have a **Year** column in all date tables (column names are unique within their table), it's not self-describing by default visual titles. Consider renaming columns in each dimension role table, so that the **Ship Date** table has a year column named **Ship Year**, etc.
- When relevant, ensure that table descriptions provide feedback to report authors (through **Fields** pane tooltips) about how filter propagation is

configured. This clarity is important when the model contains a generically named table, like **Date**, which is used to filter many fact-type tables. In the case that this table has, for example, an active relationship to the reseller sales order date column, consider providing a table description like "Filters reseller sales by order date".

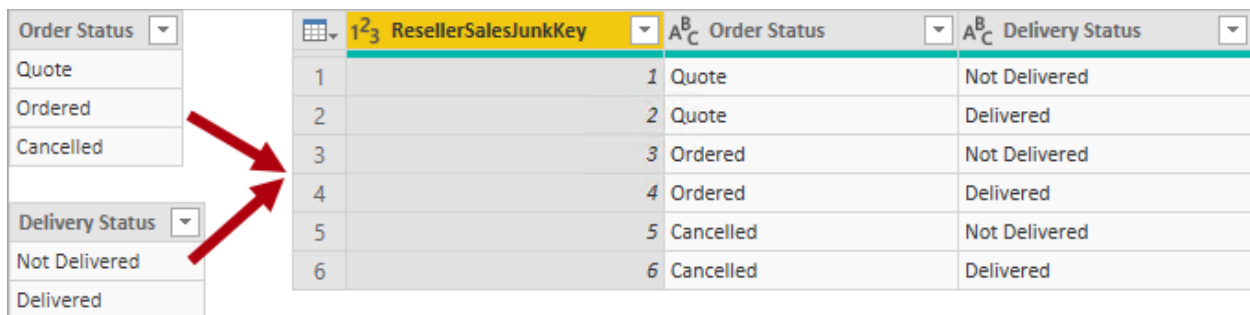
For more information, see [Active vs inactive relationship guidance](#).

Junk dimensions

A **junk dimension** is useful when there are many dimensions, especially consisting of few attributes (perhaps one), and when these attributes have few values. Good candidates include order status columns, or customer demographic columns (gender, age group, etc.).

The design objective of a junk dimension is to consolidate many "small" dimensions into a single dimension to both reduce the model storage size and also reduce **Fields** pane clutter by surfacing fewer model tables.

A junk dimension table is typically the Cartesian product of all dimension attribute members, with a surrogate key column. The surrogate key provides a unique reference to each row in the table. You can build the dimension in a data warehouse, or by using Power Query to create a query that performs [full outer query joins](#), then adds a surrogate key (index column).



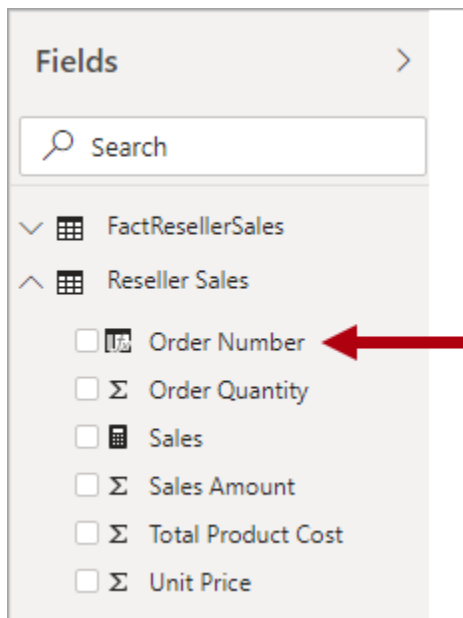
Order Status		1 ² ₃ ResellerSalesJunkKey	A ^B _C Order Status	A ^B _C Delivery Status
Quote		1	1 Quote	Not Delivered
Ordered		2	2 Quote	Delivered
Cancelled		3	3 Ordered	Not Delivered
		4	4 Ordered	Delivered
		5	5 Cancelled	Not Delivered
		6	6 Cancelled	Delivered

You load this query to the model as a dimension-type table. You also need to merge this query with the fact query, so the index column is loaded to the model to support the creation of a "one-to-many" model relationship.

Degenerate dimensions

A **degenerate dimension** refers to an attribute of the fact table that is required for filtering. At Adventure Works, the reseller sales order number is a good example. In this case, it doesn't make good model design sense to create an independent table consisting of just this one column, because it would increase the model storage size and result in **Fields** pane clutter.

In the Power BI model, it can be appropriate to add the sales order number column to the fact-type table to allow filtering or grouping by sales order number. It is an exception to the formerly introduced rule that you should not mix table types (generally, model tables should be either dimension-type or fact-type).



However, if the Adventure Works resellers sales table has order number *and* order line number columns, and they're required for filtering, a degenerate dimension table would be a good design. For more information, see [One-to-one relationship guidance \(Degenerate dimensions\)](#).

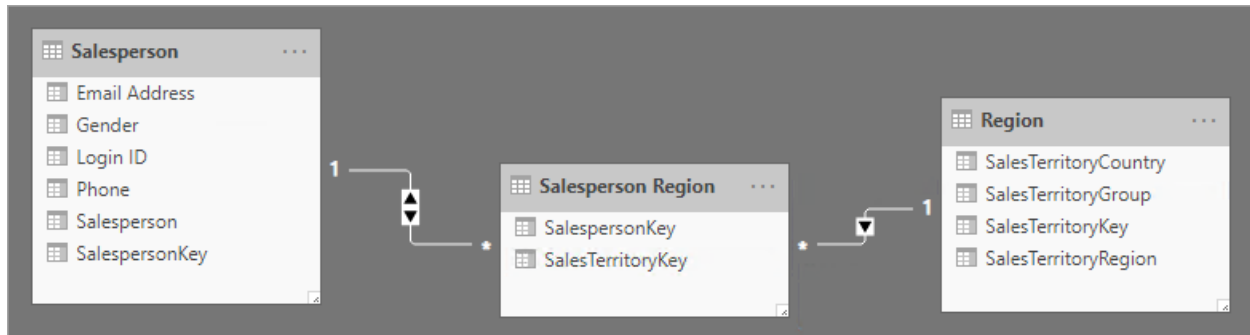
Factless fact tables

A **factless fact** table doesn't include any measure columns. It contains only dimension keys.

A factless fact table could store observations defined by dimension keys. For example, at a particular date and time, a particular customer logged into your web site. You could define a measure to count the rows of the factless fact table to perform analysis of when and how many customers have logged in.

A more compelling use of a factless fact table is to store relationships between dimensions, and it's the Power BI model design approach we recommend defining many-to-many dimension relationships. In a [many-to-many dimension relationship design](#), the factless fact table is referred to as a *bridging table*.

For example, consider that salespeople can be assigned to one *or more* sales regions. The bridging table would be designed as a factless fact table consisting of two columns: salesperson key and region key. Duplicate values can be stored in both columns.



This many-to-many design approach is well documented, and it can be achieved without a bridging table. However, the bridging table approach is considered the best practice when relating two dimensions.