O'REILLY®

Data Modeling with Microsoft Power BI

Self-Service and Enterprise DWH with Power BI

Early Release
RAW & UNEDITED

Markus Ehrenmueller-Jensen

Data Modeling with Microsoft Power BI

Self-Service and Enterprise DWH with Power BI

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Markus Ehrenmueller-Jensen

Beijing • Boston • Farnham • Sebastopol • Tokyo

Data Modeling with Microsoft Power BI

by Markus Ehrenmueller-Jensen

Copyright © 2025 O'Reilly Media. All rights reserve.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 9547.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Editors: Michelle Smith and Shira Evans
- Production Editor: Katherine Tozer
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

- October 2024: First Edition

Revision History for the Early Release

- 2023-04-14: First Release
- 2023-06-01: Second Release
- 2023-07-14: Third Release
- 2023-09-13: Fourth Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098148553 for release details.

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. *Data Modeling with Microsoft Power BI*, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher’s views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-14849-2

Chapter 1. Understanding a Data Model

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sevans@oreilly.com*.

This chapter will teach you (or refresh your memory on) the basics of data modeling. It starts with the very basic terms, but please bear with me, because it will help you to understand the reasoning behind why you should take so much care of the data model. Creating reports and doing analysis based on a data model which is optimized for these tasks is much easier, than compared to trying this with a data model optimized for other purposes (for example, a data model optimized to store data for an application or data collected in a spreadsheet). When it comes to analytical databases and data warehouses, you have more than one option. Throughout this book you will

learn that the goal is to create the data model as a *Star Schema*. At the end of this chapter, you will know which characteristics a *Star Schema* will differentiate it from other modelling approaches. With each and every chapter throughout this book you will learn more and more why a *Star Schema* is so important when it comes to analytical databases in general and Power BI and Analysis Services Tabular in particular. And I will teach you, how you can transform any data model into a *Star Schema*.

Transforming the information of your data source(s) into a *Star Schema* is usually *not an easy task*. On the opposite: It can be hard. It might take several iterations. It might take discussions with the people who you build the data model for. And the people using the reports. And the people who are responsible for the data sources. You might face doubts (from others and yourself) and might ask yourself if it is really worth all the effort instead of simply pulling the data in as it is, to avoid all the struggle. At such a point it is very important that you take a deep breath and evaluate, if a transformation would make the report creator's life easier. Because, if it will make the report creator's life easier, than it's worth the effort. Therefore, repeat the data modelers mantra together with me: *Make the report creators life easier*.

Before I actually come to talk about transformations, I will to introduce you to the basic terms and concepts in this chapter:

- What is a *data model* at all?
- What is an *entity*? What has an entity to do with a *table*?

- Why should we care about *relations*?
- I will then introduce the concept of different *keys* (*primary key*, *foreign key*, *surrogate key*) and explain the term *cardinality*.
- Next topic is how to combine tables (*set operators* and *joins*).
- And finally, we compare the different *data modeling options* and their use cases.

First stop in our journey towards the *Star Schema* is describing what a *Data Model* is in general terms.

Data Model

A model is something which *represents* the real-world. It is *not replicating* the real world. Think of a map. A map replicating 1:1 the real world would be very unpractical: It would cover the whole planet. Instead, a map scales down the distances. And a map is created for a special purpose. A hiking map will contain different information (and omit others) compared to a road map. And a nautical chart would look completely different. All of them are maps – but with different purposes.

The same applies to a data model. A data model represents a certain business logic. As with the map example, a data model for different use cases will look differently. Therefore, models for different industries will not be the same. And even organizations within the same industry will need different data models (even for basically the identical business processes), as they will

concentrate on different requirements. Throughout the book you will see several challenges and their solutions and I am confident that they will help you in overcoming the challenges you will face when building a data model for your organization.

So, the bad news is: There is not one data model, which rules them all. And: With just technical knowledge, but no domain-knowledge, it is impossible to create a useful data model. This book will guide you through the technical knowledge necessary to successfully build data models for Power BI and/or Analysis Services Tabular. But don't forget to collect all requirements from the business (or explicitly write them down, if you are the domain expert on your own) before you start with creating the data model. You can collect the requirements by writing down sentences in natural language, like "We sell goods to customers and need to know the day and the SKU of the product we sold." or "We need to analyze the 12 month rolling average of the sold quantity for each product." or "Up to ten employees form a project team and we need to report the working hours per project task." These requirements will help you to know which information you need to store in which combinations and in which level of detail. There might be more than one option for the design of a data model for a certain use case.

And there is more bad news: You need the create the correct data model right from the beginning. As soon as the first reports created on a data model, every change in the data model bears the risk of breaking those reports. The later you discover inconsistences and mistakes in your data model, the more

expensive it will be to correct them (in terms of time to check and correct all depending reports). This cost hits everybody who created those reports: Yourself, but also other users who built reports based upon your data model.

The design of the data model has a huge impact on the performance of your reports, which query the data from the data model to feed the visualizations, as well. A well-designed data model lessens the need for query tuning later. And a well-designed data model can be used intuitively by the report creators - saving them time and effort (and therefore saving your organization money). In a different point of view: Problems with the performance of a report or report-creators unsure of which tables and columns they need to use for to gain certain insights are a sure sign for a data model which can be improved by a better choice of design.

Later, in section “Entity-Relationship Diagrams (ERD)” I will describe graphical ways of how to document the shape of a data model. Immediately in the next section, we will learn that a data model consists of entities.

Basic Components

Before you dive in, there are a few key components of a data model that you will need to understand. In the next section, I will explain the basic parts of a data model. Later I will walk you through different ways of combining tables with the help of set operators and joins, and which kind of problems you can

face and how to solve them.

Entity

An entity is someone or something which can be individually identified. In natural language entities are nouns. Think of a real person (your favorite teacher, for example), a product you bought recently (ice cream, anybody?) or a term (e. g. “entity”).

Entities can be both, real and fictious. And most entities have attributes: a name, a value, a category, point in time of creation, etc. These attributes are the information we are after. These attributes are displayed in reports to help the reader of the reports to provide context, gain insights, and get to decisions. They are used to filter displayed information to narrow down an analysis, too.

Maybe you wonder, how such entities make it into a data model? They are stored in tables, as you find out in the next section.

Tables

Tables are the base of a data model. They are part of data models at least since around 1970, when Edgar F. Codd developed the relational data model for his employer, IBM. But I am convinced, that collecting information as lists or tables was already done way before the invention of computers, as

you can see when looking old books. Tables are hosting entities: Every entity is represented by a row in a table. Every attribute of an entity is represented by a column in a table. A column in a table has a name (e. g. birthday) and a data type (e. g. date). All rows of a single column must conform to this data type (it is not possible to store the place of birth in the column birthday for any row). This is an important difference between a (database's) table and a spreadsheet's worksheet (e. g. in Excel). A single column in a single table contains content.

Table 1-1. A table containing the name of doctors

Doctor's Name	Hire Date
Smith	1993-06-03
Grey	2005-03-27
Young	2004-12-01
Stevens	2000-06-07
Karev	1998-09-19
O'Malley	2003-02-14

Entities do not exist just on their own, but are related to each other. This is what I describe in the upcoming section.

Relations

Relations connect (in most cases) only two entities. In natural language the relationship is represented by a verb (e. g. bought). Between the same two entities there might exist more than one single relationship. For example, a customer might have first ordered a certain product, which we later shipped. It's the same customer and the same product, but different relations (ordered vs. shipped).

Some relationships can be self-referencing. That means, that there can be a relationship between one entity (= one row in this table) and another entity of the same type (= a different row in the same table). Organizational hierarchies a are a typical example here. Every employee (except maybe the CEO) needs to report to her boss. The reference to the boss is therefore an attribute. One column contains the identifier of the employee (e. g. `Employee ID`) and a different column contains the identifier of who this employee reports to (e. g. `Manager ID`). The content of `Manager ID` of one row can be found as the `Employee ID` of a different row in the same table.

Examples for relations express in natural language:

- Dr. Smith *treats* Mr. Jones
- Michael *attended* (course) “Data Modeling”
- Mr. Gates *owns* Microsoft
- Mr. Nadella *is* CEO

When you start collecting the requirements for a report (and therefore for the necessary content of your data model) it makes sense to write everything down in sentences in natural language, as in the examples here. This is the first step. Later you will learn that you can also draw tables and their relationships as an “Entity-Relationship Diagrams (ERD)”.

Sometimes the existence of a relationships alone is enough information to collect and satisfy analysis. But some relationships might have attributes, which we collect for more in-depts analysis:

- Dr. Smith treats Mr. Jones *against flue*
- Michael attended (course) “Data Modeling” *with grade A*
- Mr. Gates owns Microsoft *to 50%*
- Mr. Nadella is CEO *since February 4 2014*

You learned that entities are represented as rows in tables. The question remains, how you can then connect these rows with each other to represent their relationship. The first step to the solution is, to find a (combination of) columns, which will uniquely identify a row. Such a unique identifier is called a *Primary Key* and you will learn all about it in the next section.

Primary Keys

Both, in the real world and in a data model, it is important, that we can uniquely identify a certain entity (= row in a table). People, for example, are identified via their names in the real world. When we know two people of the same (first) name, we might add something to their name (e. g. their last name) or invent a call name (which is usually shorter than the first name and last name combined), so that we can make clear, who we are referring to (but don’t spend too much time). If we don’t pay attention, we might end up with a confusing conversation, where one is referring to one person and the other to a different person (“Ah, you are talking about the other John!”).

In a table it is very similar: We can mark one column (or a combined set of columns, which is called a composite key) as the primary key of the table. If we do not do that, we might end up with confusing reports because of duplicates (as the combined sales of both Johns are shown for every John). Best practice is, to have a single column as a primary key (as opposed to composite keys) for the same reason I use call names for people: because it is shorter and therefore easier to use. You can only define one single primary key.

Explicitly defining a primary key (short: PK) on a table has several consequences: It puts a unique constraint on the column (which guarantees, that no other row can have the same value(s) in the primary key; rejecting both, inserts and updates, which would violate this rule). Every relational

database management system I know, also puts an index on the primary key (to speed up the lookup for, if an insert or update would violate the primary key). And all columns used as a primary key must contain a value (nullability is disabled). I strongly believe that you should have a primary key constraint on every table.

In a table we can make sure, that every row is uniquely identifiable, by marking one or the combinations of several rows as the primary key. To make the example with `Employee ID` and `Manger ID` work, it is crucial that the content of column `Employee ID` is unique for the whole table.

Typically, in a data warehouse (= a database built for the sole purpose of making reporting easier) you would not use one of the columns of the source system as a primary key (e. g. first name & last name or social security number), but introduce a new artificial ID, which only exists inside the data warehouse: a surrogate key (s. “Surrogate Keys”).

Surrogate Keys

A *Surrogate Key* is an artificial value, only created in the data warehouse. It is neither an entity’s natural key nor the business key of the source system, and definitely not a composite key (but a single value). It is created solely for the purpose of having one single key column, which is independent of any source system. Think of it as a (weird) call name.

Typically, the columns have “ID”, “SID”, “Key”, etc. as part of their names. The common relational database management system are able to automatically find a value for this column for you. Best practice is, to use an integer value, starting at 1. Depending on the number of rows you expect inside the table, you should find the appropriate type of integer, which usually can cover something between 1 Byte (= 8 bit = 2^8 = 256 values) and 8 Bytes (= 8 * 8bits = 64 Bits = 2^{64} = 18 446 744 073 709 551 616 values). Sometimes global uniqueidentifiers are used. They have their use case in scale-out scenarios, where processes need to run independently from each other, but still generate surrogate keys for a common table. They require more space to store, compared to an integer value. That’s why I would only use them in cases when integer values can absolutely not be used.

The goal is to make the data warehouse independent from changes in the source system (e. g. when an ERP system is changed and it might re-use IDs for new entities or when the data type of the ERP’s business key changes).

Surrogate Keys are also necessary, when you want to implement a *Slowly Changing Dimension Type 2*, about which we will talk in a later chapter.

I still did not explain, how you can represent relations of entities. Now it is time for that. The solution is not very complicated: You just store the *Primary Key* of an entity as column in an entity who has a relationship to it. This way you reference another entity. You reference the *Primary Key* of a *foreign* entity. That’s why this column is called a *Foreign Key*.

Foreign Keys

Foreign keys are simply referencing a primary key (of a foreign entity). The primary key is hosted in a different column, either in a different table, or the same table. For example, the sales table will contain a column `Product ID` to identify, which product was sold. The `Product ID` is the primary key of the `Product` table. On the other hand, the `Manger ID` column of the `Employee` table refers to column `Employee ID` in the very same table (`Employee`).

When we explicitly define a foreign key constraint on the table, the database management system will make sure, that the value of the foreign key column for every single row can be found as a value in the referred primary key. It will guarantee, that no insert or update in the referring table can change the value of the foreign key to something invalid. And it will also guarantee, that a referred primary key can not be updated to something different or deleted in the referred table.

Best practice is to disable nullability for a foreign key column. If the foreign key value is (yet) not known or does not make sense in the current context, than a replacement value should be used (typically surrogate key -1). To make this work, you need to explicitely add a row with -1 as it’s primary key to the referred table. This gives us better control of what to show in case of a missing value (instead of just showing an empty value in the report). It also allows for inner joins, which are more performant compared to outer joins

(which are necessary when a foreign key contains null to not loose those rows in the result set; read more about “Joins”).

While creating primary key constraints will automatically put an index on the key, creating foreign key constraints is not implemented this way in e. g. Azure SQL DB or SQL Server. To speed up joins between the table containing the foreign key and the table containing the primary key, indexing the foreign key column is strongly recommended.

A next question might arise in your mind: How do I know, in which of two entities involved in a relationship do I store the *Foreign Key*? Before I can answer this question, you need first to understand that there are different types of relationships. These types are described as the *Cardinality* of a relationship.

Cardinality

Cardinality describes, how many rows can (maximally) be found for a given row in a related table. For two given tables, the cardinality can be one of the following ones:

- One-to-many (1:m, 1-*)
  For example: One customer may have many orders. One order is from exactly one customer.
- One-to-one (1:1, 1-1)

For example: This person is married with this other person.

- Many-to-many (m:m, *-*)
  For example: One employee works for many different projects. One project has many employees.

If you want to be more specific, you can also describe, if a relationship can be conditional. As all relationships on the “many” side are conditional (e. g. a specific customer might not have ordered yet) this is usually not explicitly mentioned. Relationships on the “one” side could be conditional (e. g. not every person is married). We might then change the relationship description from 1:1 to c:c in our documentation.

Combining Tables

So far you learned that information (= entities and their relationships) is stored in tables in a data model. Before I introduce you to rules, when to split information into different table or keep it together in on single table, I want to discuss how we can combine information spread into different tables in the upcoming section.

Set Operators

You can imagine a “set” as the result of a query, or as rows of data in a tabular shape. *Set operators* allow us to combine two (or more) query results, by adding or removing rows. It’s important to keep in mind that the number

of columns in the queries involved must be the same. And the data types of the columns must be identical or the data type conversion rules of the database management you are using are be able to (implicitly) convert to the data type of the column of the first query. A *set operator* does not change the number or type of columns, only the number of rows. “Set Operators” is a graphical representation of the following explanation:

Union

Adds the rows from the second set to the rows of the first set. Depending on the implementation, duplicates may appear in the result or be removed by the operator. For example, you want a combined list of both, customers, and suppliers.

Intersect

Looks for rows, which appear in both sets. Only rows appearing in both sets are kept, all other rows are omitted. For example, you want to find out, who appears to be both, a customer, and a supplier, in your system.

Except (or minus)

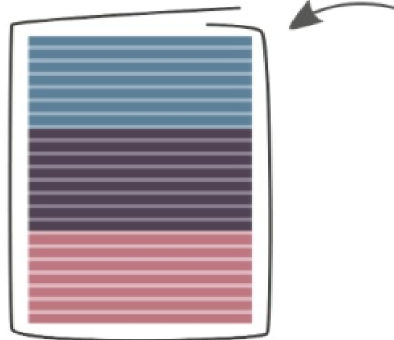
Looks for rows, which appear in both sets. Only rows from the first set, which are not appearing in the second set are returned. You “subtract” the rows of the second table from the rows of the first table (hence, this operator is also called minus). For example, you want to get a list of customers, limited to those, who are not also a supplier.

WARNING

The comparison, if a row is identical or not is done by evaluating and comparing the content of all columns of the row of the query. Pay attention here, as while the primary keys listed in the result set might be identical, the names or description might be different. Rows with identical keys but different descriptions will not be recognized as identical by a *set operator*.

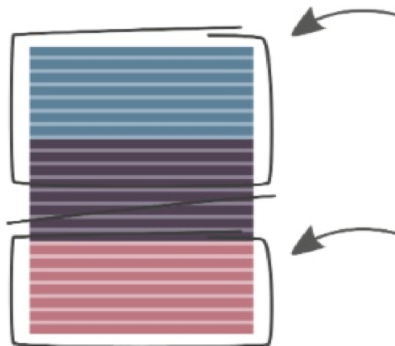
UNION ALL

1	A
2	B
2	B
3	C



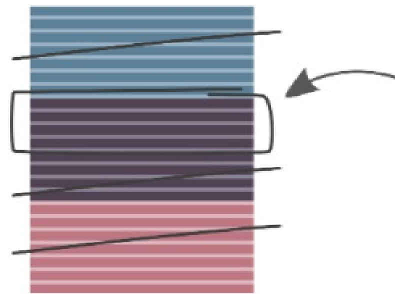
UNION

1	A
2	B
~~2~~	~~B~~
3	C



INTERSECT

~~1~~	~~A~~
2	B
~~2~~	~~B~~
~~3~~	~~C~~



EXCEPT

1	A
~~2~~	~~B~~
~~2~~	~~B~~
~~3~~	~~C~~

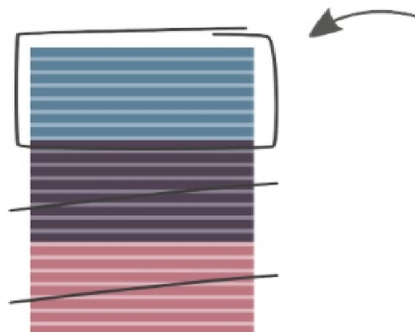


Figure 1-1. Set Operators

As you learned, *set operators* are combining tables in a vertical fashion: They basically append the content of one table to the content of another table. The number of columns can not change with a *set operator*. If you need to combine tables in a way where you add columns of one table to the columns of another table, you need to work with *join operators*.

Joins

Joins are like the set operators in the sense that they are also combining two (or more) queries (or tables). Depending on the type of the join operator, you might end up with the same number of rows of the first table, or more or less rows. With joins you can also add columns to a query (which you can not do with a *set operator*).

While set operators compare all columns, joins are done on only a selected (sub-)set of columns, which you need to specify (in the so-called *join predicate*). For the *join predicate* usually you will use an equality comparison between the primary key of one table and the foreign key in the other table (*equi join*). For example, you want to show the name of a customer for a certain order (and specify an equality comparison between the order table's foreign key `Customer Key` and the customer table's primary key `Customer Key` in the join predicate).

In only special cases you would compare other (non-key) columns with each other to join two tables. You will see examples for such joins in the chapters about use cases, where I will pick advanced problems and demonstrate their solution. There you will also use comparisons done not on the equality of two columns, but use comparison operators like between, greater equal, not equal, etc. (such joins are called *non-equi join*). One of the examples is about grouping values (= binning). This is about finding the group in which a certain value falls into by joining the table containing the groups with a condition asking for that the value is greater or equal than the lower range of the bin and lower than the upper range of the bin. While the range values form the composite primary key of the table containing the groups, the lookup value is not a foreign key: It is an arbitrary value, possible not found as a value in the lookup table, as the lookup table only contains a start and end value per bin, but not all the values within the bin.

Natural joins are a special case of equi-joins. In such a join you do not specify the columns to compare. The columns to use for the equi-joins are automatically chosen for you: columns with the same name in the two joined tables are used. As you might guess, this only works, if you stick to a naming convention (which is a very good idea anyways) to support these joins.

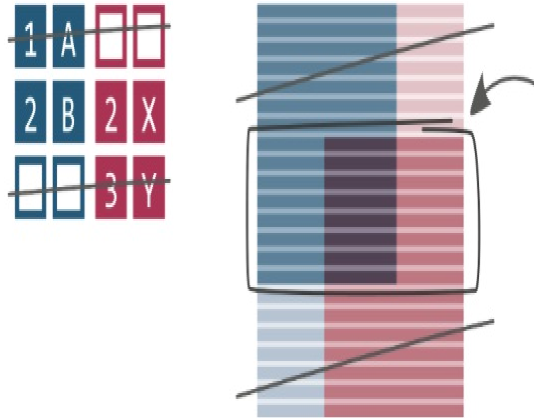
Earlier in this chapter I used `Employee` as an typical example, where the foreign key (`Manager Key`) references a row in the same table (via primary key `Employee Key`). If you actually join the `Employee` table with itself, to find e. g. the manager’s name for an employee, you are implementing a

self join.

The important difference between set operators and joins is, that joins are adding *columns* to the first table, while set operators are adding *rows*. Joins allow you to add a category column to your products which can be found in a lookup table. Set operators allow you to combine e. g. tables containing sales from different data sources into one unified sales table. So, I imagine set operators as a combination of tables in a vertical manner (putting two tables underneath each other). And join operators as a combination of tables in a horizontal manner (putting two table side-by-side). This mental concept is not exact in all regards (set operators `INTERSECT` and `EXCECPT` will remove rows and joins will also add or remove rows depending on the cardinality of the relationship or the join type) but it is, as I think, a good starting point to differentiate both.

We can join two tables in the following manners:

Inner join
Looks for rows, which appear in both tables. Only rows appearing in both tables are kept, all other rows are omitted. For example, you want to get a list of customers for which we can find an order. You can see a graphical representation in Figure 1-2.



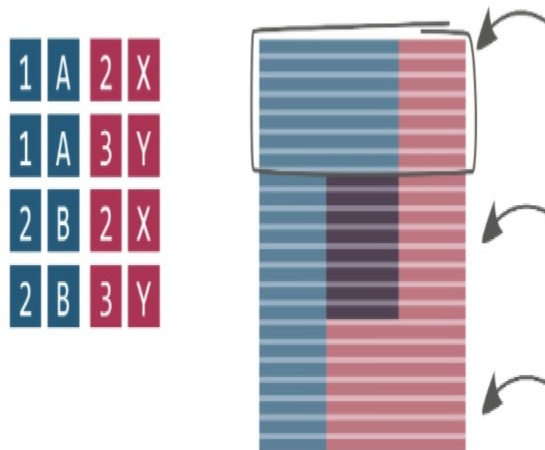


Figure 1-2. Inner join and outer join

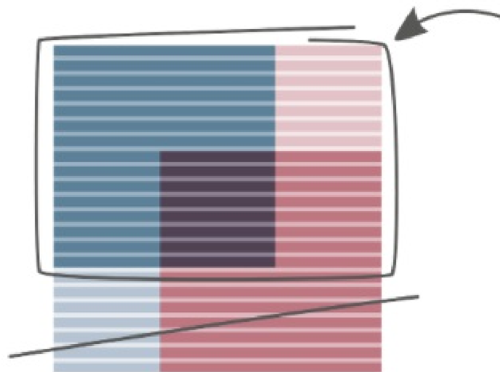
This is similar to the `INTERSECT` set operator. But the result can contain

the same, more or less rows than the first table contains. It will contain the same number of rows, if for every row of the first table exact one single row in the second table exists (e. g. when every customer has placed exactly one single order). It will contain more rows, if there is more than one matching row in the second table (e. g. when every customer has placed at least one order or some customers have so many orders, that they make up for other customers who didn't place any order). It will contain less rows, if some rows of the first table can't be matched to rows in the second table (e. g. when not all customers have placed orders and these missing orders are not compensated by other customers). The latter is the "danger" of inner joins: The result may skip some rows of one of the tables (e. g. the result will not contain rows for customers without orders).

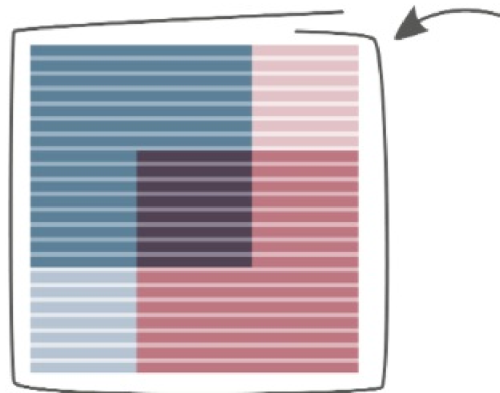
Outer join

Returns all the rows from one table and values for the columns of the other table from matching rows. If no matching row can be found, the value for the columns of the other table are *null* (and the row of the first table is still kept). This is shown in a graphical manner in Figure 1-3.

LEFT (OUTER) JOIN:



FULL (OUTER) JOIN:



RIGHT (OUTER) JOIN:

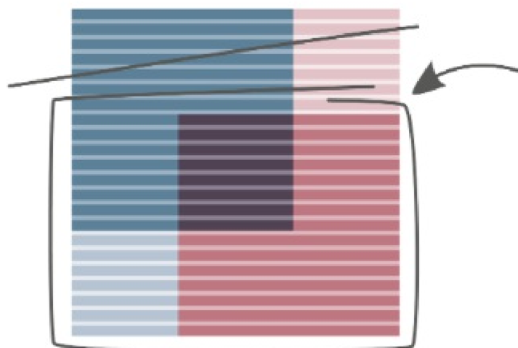


Figure 1-3. Outer join

You can ask for either all rows of the first table in a chain of join operators (left join), making the values of the second table optional. Or the other way around (right join). A full outer join makes sure to return all rows from both tables (with optional values from the other table).

For example, you want a list of all customers with their order sales from the current year, even when the customer did not order anything in the current year (and then display null or 0 as their order sales).

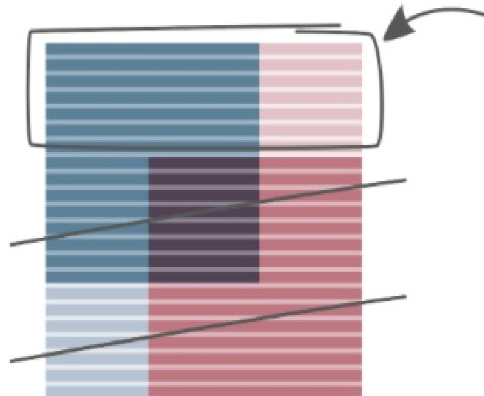
There is no similar set operator to achieve this. An outer join will have at least so many rows as an inner join. It’s not possible that an outer join (with the identical join predicated) returns less rows than an inner join. Depending on the cardinality it might return the same number of rows (if there is a matching row in the second table for every row in the first table) or more (if some rows of the first table cannot be matched with rows of the second table, which are omitted by an inner join).

Anti-join

An anti-join is based on an outer join, where you only keep the rows not existing in the other table. The same ideas for left, right and full apply here, as you can see in Figure 1-4.

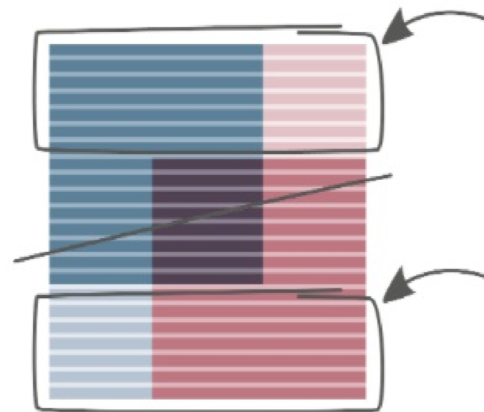
LEFT ANTI-JOIN:

1	A		
~~2~~	~~B~~	~~2~~	~~X~~
		~~3~~	~~Y~~



FULL ANTI-JOIN:

1	A		
~~2~~	~~B~~	~~2~~	~~X~~
		3	Y



RIGHT ANTI-JOIN:

~~1~~	~~A~~		
~~2~~	~~B~~	~~2~~	~~X~~
		3	Y

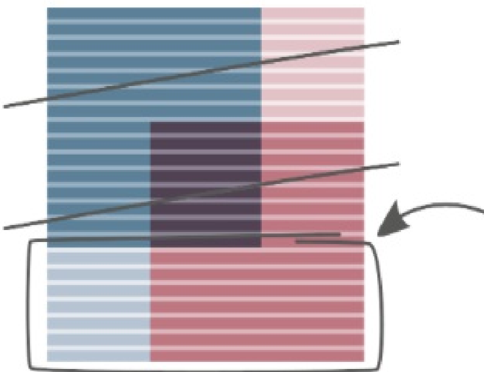


Figure 1-4. Anti join

For example, you want a list of customers, who did not order anything in the current year (to send them an offer they can’t refuse).

There is no similar set operator to achieve this. The anti-join delivers the difference of an inner join compared to an outer join.

Cross Join

Creates a so-called cartesian product. Every single row of the first table is combined with each and every row from the second table. In many scenarios this does not make sense (e. g. combining every row of the sales table with every customer, independent, if the row of the sales table is for the customer or a different one). Practically you can create queries, which show thinkable combinations. For example, by applying a cross join on the sizes of clothes with all the colors, you get a list of all thinkable combinations of sizes and colors (independent, if a product really is available in this combination of size and color). A cross join can be a basis for a left join or anti-join, to show explicitly point out combinations with no values available. You can see an example of the result of a crosss join in Figure 1-2

Do you feel dizzy because of all the different join options? Unfortunately, I need to add one layer of complexity in the next section. As you just have learned, when joining two tables, the number of rows in the result set might

be smaller, equal or higher than the number of rows of a single table involved in the operation. The exact number depends on both, the type of the join and the cardinality of the tables. In a chain of joins involving several tables, the combined result might lead to undesired results, as you will learn in the next section.

Join Path Problems

When you join the rows of one table to the rows of another table you can face several problems, resulting in unwanted query results. The possible problems are:

- Loop
- Fan Trap
- Chasm Trap

Let's take a closer look onto them:

Loop
You face this problem in a data model if there is more than one single path between two tables. It does not have to be a literal loop in your entity-relationship diagram, where you can "walk" a join path in a manner where you return to the first table. We speak already of a loop, when a data model is ambiguous. And this cannot only exist in very complex data models, but also in the very simple setting of having just more than one

direct relationship between the same two tables. Think of a sales table containing an order date and a ship data column (Figure 1-5). Both columns have a relationship to the data column of the date table. If you join the date table to the sales table on both, the order date and the ship date, you will end up with only sales, which were ordered and shipped on the very same day. Sales, which were ordered and shipped on different days, will not show up as a result for your query. This might be an unexpected behavior, returning too few rows.

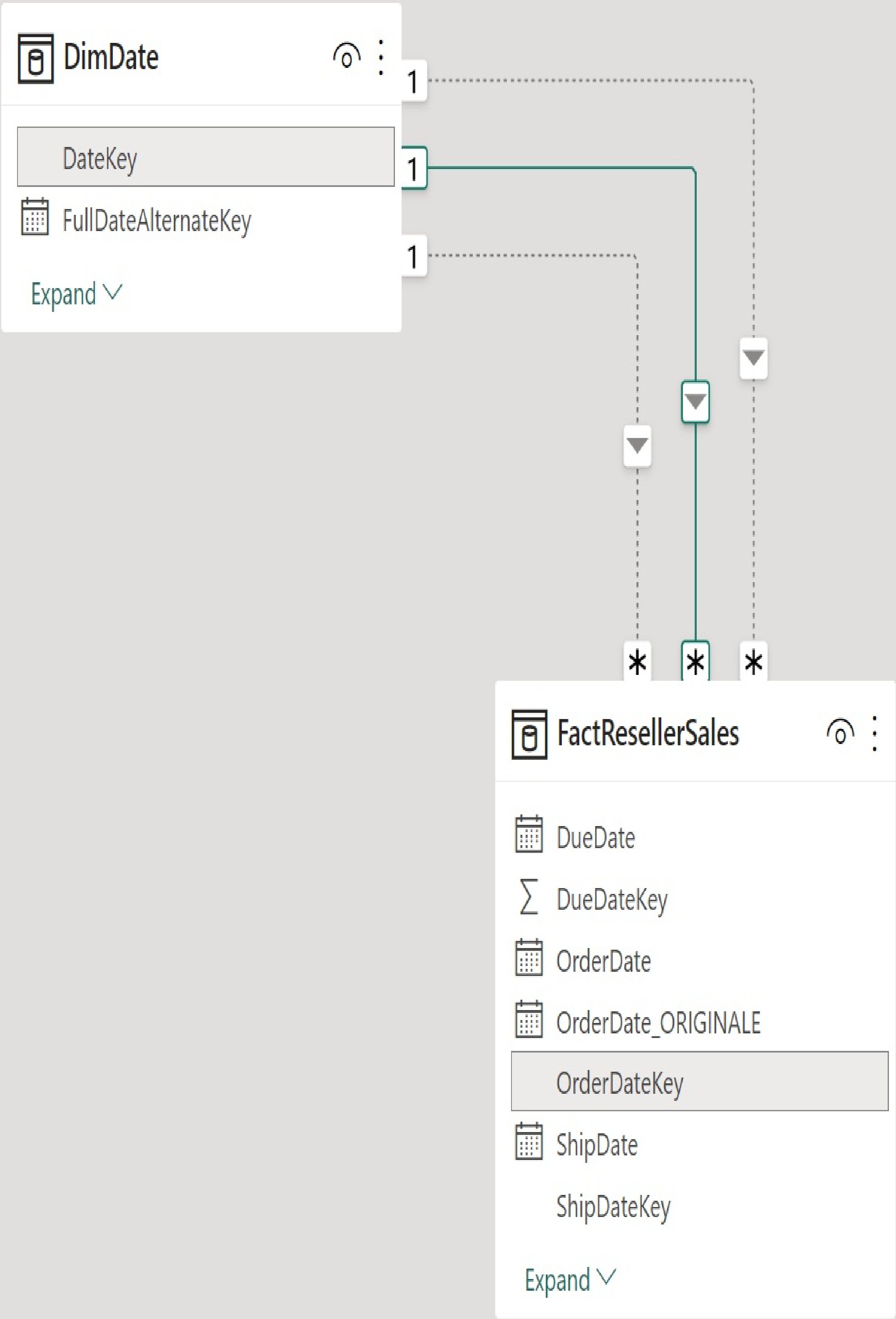

DimDate
DateKey
FullDateAlternateKey
Expand
1
1
1
*
*
*
FactResellerSales
DueDate
DueDateKey
OrderDate
OrderDate_ORIGINALE
OrderDateKey
ShipDate
ShipDateKey
Expand

Figure 1-5. Join Path Problem: Loop

The cure against a loop is to (physically or logically) duplicating the date table and joining one date table on the order date and the other date table on the ship date.

Chasm trap

The chasm trap (s. Figure 1-6) describes a situation in a data model, where you have a converging many-to-one-to-many relationship. For example, you could store the sales you are making over the internet in a different table than the sales you are making through resellers. Both tables can though be filtered over a common table, let’s say, a date table. The date table has a one-to-many relationship to each of the two sales tables – creating a many-to-one-to-many relationship between the two sales tables.

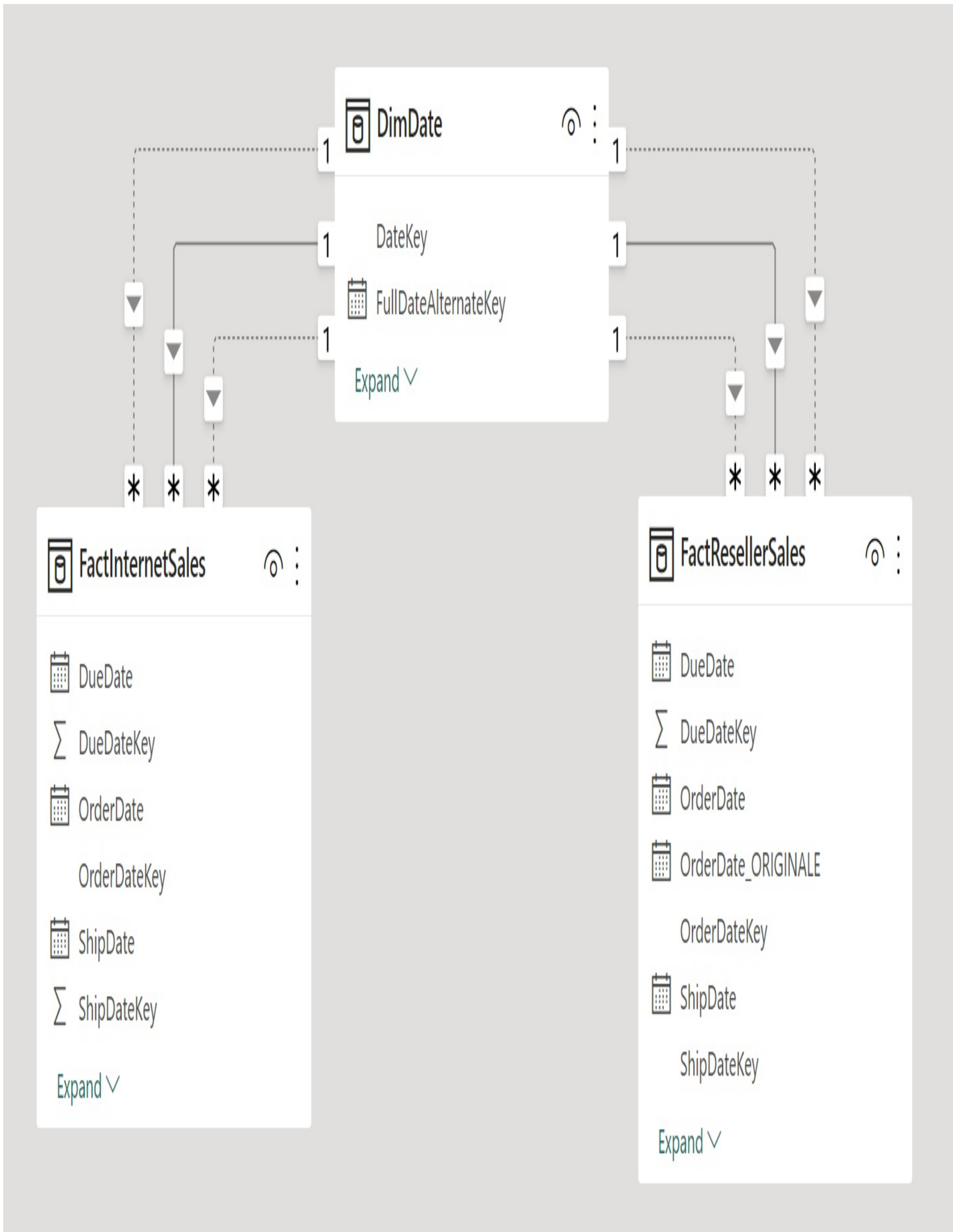


Figure 1-6. Join Path Problem: Chasm Trap

When you have more than one sale over the internet for a particular day, you would join the same row of the date table multiple times. As there is also a chance, that for this day you also sold several items to resellers, every single internet sale will then be duplicated per row in the reseller sales table. Your query would then report too high numbers of internet sales and too high numbers of reseller sales.

The cure against the chasm trap problem depends on the tool you are using. Jump to the chapters in the other parts of this book to read how you solve this in DAX, Power Query / M and SQL.

Fan trap

You can step into a fan trap (Figure 1-7) in situations, where you want to aggregate on a value on the one-side of a relationship, while joining a table on the many side of the same relationship. For example, you could store the freight cost in a sales header table, which holds information per order. When you join this table with the sales detail table, which holds information per ordered item of the order (which could be multiple per order), you are duplicating the rows from the header table in the query result, therefore duplicating the amount of freight.

Sales SalesOrderHead...

AccountNumber
∑ BillToAddressID
Comment
CreditCardApprovalCode
∑ CreditCardID
∑ CurrencyRateID
∑ CustomerID
DueDate
∑ Freight

Collapse

1

*

Sales SalesOrderDetail

ModifiedDate
SalesOrderID

Expand

Figure 1-7. Join Path Problem: Fan trap

> The cure against the fan trap problem depends on the tool you are using, too. Jump to the chapters in the other parts of this book to read how you solve this in DAX, Power Query / M and SQL.

As you saw in the screenshots, drawing the tables and the cardinality of their relationships can help in getting an overview about potential problems. The saying “A picture says more than a thousand words.” applies to data models as well. I introduce such *Entity-Relationship Diagrams* in the next section.

Entity-Relationship Diagrams (ERD)

An Entity-Relationship Diagram, or short ERD, is a graphical representation of entities and the cardinality of their relationships. When a relationship contains an attribute, it might be shown as a property of the relationship as well. Over the years different notations have been developed (*https://www.lucidchart.com/pages/er-diagrams* contains a nice overview about the most common notations). In my point of view, it is not so important which notation you are using – it’s more important to have an ERD at hand for your whole data model. If the data model is very complex (= contains a lot of tables) it is common to split it into sections, sub-ERDs.

Deciding on the cardinality of a relationship and documenting it (e. g. in the form of an ERD) will help to find out, in which table you need to create the

foreign key. Look at the following examples:

The cardinality of the relationship between customers and their orders should be a one-to-many relationship. One customer can possibly have many orders (even when some customers only have a single order or others don’t have any order yet). On the other hand, a particular order is associated with one single customer only. This knowledge helps us to decide, if we need to create a foreign key in the customer table to refer the primary key of the order table, or the other way around. If the customer table contains the `Order Key`, it will allow each customer to refer to a single order only. And, any order, could be referenced by multiple customers. So, plainly, this approach would not reflect the reality in a correct manner. That’s why we need a `Customer Key` (as a foreign key) in the order table instead, as shown in Figure 1-8. Then, every row in the order table, can only refer a single customer. And a customer can be referenced by many orders.

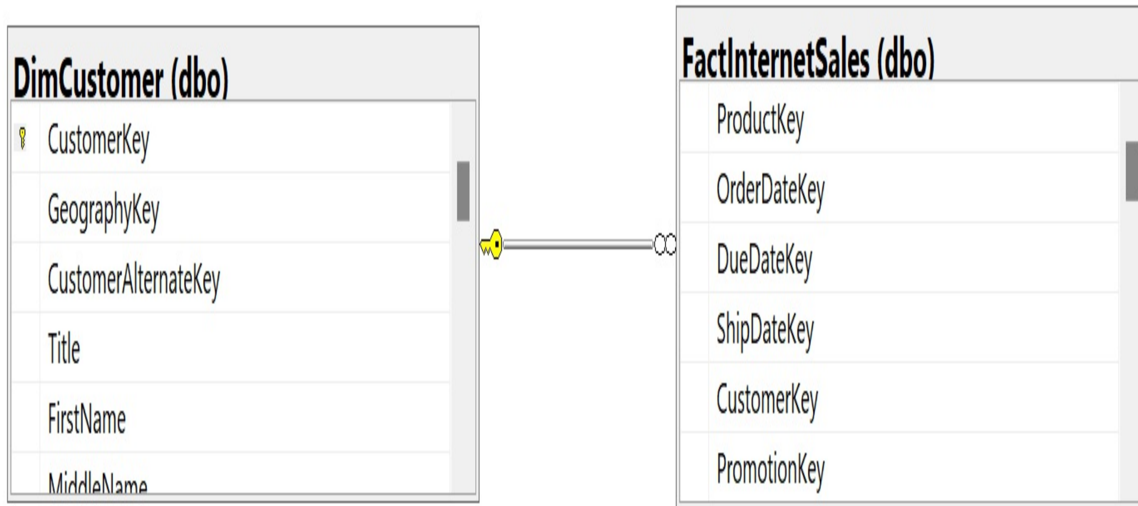


Figure 1-8. Entity-Relationship Diagramm for tables customer and orders

In case that an order could be associated with more than a single customer, we would face a many-to-many relationship. (As a customer could still have more than one order.) Many-to-many relationships are typical if you want to find a data model to represent employees and in which projects they are engaged. Or collecting the reasons for a sale from your customers. The same reason will be given for more than one sale. And a customer would tell you several reasons, why she made the sale.

Typically, we would add a foreign key to neither the sales table nor the sales reason table, but create a new table on its own, consisting of a composite primary key: the primary key of the sales table (`SalesOrderNumber` and `SalesOrderLineNumber` in our example, shown in Figure 1-9) and the

primary key of the sales reason table (`SalesReasonKey`). This new table has a many-to-one relationship to the sales table (over the sales' table primary key) and a many-to-one relationship to the sales reason table table (over the sales reason's table primary key). It's therefore called a bridge table, as it is bridging the many-to-many relationship between the two tables and converting it into two one-to-many relationships.

FactInternetSales (dbo)

- ProductKey
- OrderDateKey
- DueDateKey
- ShipDateKey
- CustomerKey
- PromotionKey
- CurrencyKey
- SalesTerritoryKey
- SalesOrderNumber
- SalesOrderLineNumber
- RevisionNumber
- OrderQuantity
- UnitPrice
- ExtendedAmount
- UnitPriceDiscountPct
- DiscountAmount
- ProductStandardCost
- TotalProductCost
- SalesAmount
- TaxAmt
- Freight
- CarrierTrackingNumber
- CustomerPONumber
- OrderDate
- DueDate
- ShipDate

FactInternetSalesReason (dbo)

- SalesOrderNumber
- SalesOrderLineNumber
- SalesReasonKey

DimSalesReason (dbo)

- SalesReasonKey
- SalesReasonAlternateKey
- SalesReasonName
- SalesReasonReasonType

Figure 1-9. Entity-Relationship Diagramm for tables sales and sales reason

In the other parts of this book, you will learn about practical ways of creating ERD for your (existing) data models.

Data Modeling Options

By now you should have a good understanding of the moving parts of a data model. Therefore, it is about time to talk about the different options of how to spread information over tables and relationships in a data model. This is, what the next sections will teach you.

Types of Tables

Basically, you can assign each table in your data model to either of three types:

Entity table
Rows in such a table represent events in the real world. These tables are also referred to as business entity, data, detail or fact tables.

Examples include orders, invoices, etc.

Lookup table
They are used to store more detailed information, which you do not want

to repeat in every row of the entity table. These tables are also referred to as main data or dimension.

Examples include customer, product, etc.

Bridge table
A bridge table dissolves a single many-to-many relation ship to two one-to-many relationships. In many database systems, two one-to-many relationships can be handled more gracefully than one many-to-many relationship.

For example, to link a table containing all employees and a table containing all projects.

Maybe you do not want to split your data into tables but keep it in one single table. In the next section I will describe the pros and cons of such an idea.

A Single Table To Store It All

Having all necessary information in one single table has its advantages: It is easy to read by humans. And therefore, it seems to be a natural way of storing and providing information. If you take a random Excel-File, it will probably contain one (or more tables) and all relevant information is listed as columns per single table. Excel even provides you with functions (e. g. VLOOKUP) to fetch data from a different table to make all necessary information available at one glance. Some tools (e. g. Power BI Report Builder, with which you

create paginated reports) require you to collect all information into one single query, before you can start building a report. If you have a table containing all the necessary information, writing this query is easy, as no joins are involved.

Power BI Desktop and Analysis Services Tabular are not those tools. They require you to create a proper data model. And a data model always consists of more than one single table. In the next section you will learn rules, how to create several tables to achieve the goal of a redundancy-free data model.

Normal Forms

The term *normalizing* in the context of databases was introduced by Edgar F. Codd, the inventor of relational databases. Personally, I don’t like the term very much (as I think it is a confusing term, as it’s hard to tell, what’s normal and what’s not, if you think about your real life in general, and database in particular). But I like the idea and the concept behind this term very much: The ultimate goal of normalizing a database is to remove redundancy, which is a good idea in many situations.

If you would store the name, address, email, phone, etc. of the customer in each and every order, you would store this information redundant. On one hand, you would use more storage space as necessary, due to the duplicated information.

On the other hand, this makes changes to the data over-complicated. You would need to touch not just a single row for the customer, but many (in the combined order table) if the content of an attribute changes. If the address of a customer changes, you would need to make sure, to change all occurrences of this information over multiple rows. If you want to insert a new customer, who just registered in our system, but didn't order something yet, you have to think of which placeholder value to store in the columns which contain the order information (order number, amount, price, etc.) until an order is placed. If you delete an order, you have to pay attention, to not accidently also remove the customer information, in case this was the customer's only order.

To normalize a database, you apply a set of rules, to bring it from one state, to the other. Here I will show you the rules to bring a database into *third normalform*, which is the most common normalform. If you want to dig deeper, you will find books explaining the Boyce-Codd-Normalform, fourth and fifth normalform, which I consider mainly as academic, and less practical relevant.

First Normalform (1NF)
You need to define a primary key and remove repeating column values.

Second Normalform (2NF)
Non-key columns are fully dependent on the primary key.

Third Normalform (3NF)

All attributes are directly dependent on the primary key.

The following sentence helps me to memorize the different rules: *Each attribute is placed in an entity where it is dependent on the key, the whole key, and nothing but the key … so help me, Codd.*

Let's apply those rules on a concrete example:

Table 1-2. A table violating the rules of normalization

StudentNr	Mentor	MentorRoom	Course1
1022	Jones	412	101-07
4123	Smith	216	201-01

This is one single table, containing all the necessary information. In some situations, such a table is very useful, as laid out in "A Single Table To Store It All". But this is not a data model. And it clearly violates all three rules of normalization.

In this example we see that there are repeating columns for the courses a student attends (columns `Course1`, `Course2`, and `Course3`). Sometimes information is not split into several columns, but all information is stored in a single column, separated by commas, or stored as JSON or XML in a single text column would (e. g. think of a list of phone numbers). These examples

violate the rule of the first normalform, as well. We need to deserialize the information, and split the information into rows instead, so that we get one single column with the content split out into separate rows. This transforms the table towards the first normal form (*1NF*).

Moreover, we need to find a primary key, which uniquely identifies every row of this new table. In this first step we do not introduce a new (surrogate) key but can live with a composite primary key. The columns, which make up the primary key are printed in `monospaced` (`StudentNr` and `Course`).

Table 1-3. A table in first normalform (with a composite primary key consisting of `StudentNr` and `Course`)

`StudentNr`	**Mentor**	**MentorRoom**	`Course`
1022	Jones	412	101-07
1022	Jones	412	143-01
1022	Jones	412	159-02
4123	Smith	216	201-01
4123	Smith	216	211-02
4123	Smith	216	214-01

To transform this table into second normalform (*2NF*), we start at a table in first normalform (*1NF*) and have to guarantee, that all columns are *functional dependent* on (all columns of) the primary key. A column is *functional dependent* on the primary key if a change in the content of the primary key also requires a change in the content of the column. A look on the table makes it clear, that the column `Mentor` is functional dependent on the column `StudentNr`, but apparently not on the column `Course`. No matter which courses a student attends, his or her mentor stays the same. Mentors are assigned to students in general, not on a by-course basis. And the same applies to the column `MentorRoom`. So, we can safely state, that columns `Mentor` and `MentorRoom` are functional dependent on only the `StudentNr`, but not on `Course`. Therefore, the current design violates the rules for second normalform. To achieve the second normalform, we have to split the table into two tables. One containing columns `StudentNr`, `Mentor`, and `MentorRoom +(with +StudentNr +as its single primary key). A second one, containing +StudentNr` and `Course`, only. Both columns form the primary key of this table.

Table 1-4. Table `Student` in second normalform (with primary key `StudentNr`)

`StudentNr`	**Mentor**	**MentorRoom**
1022	Jones	412

4123	Smith	216

Table 1-5. Table `StudentCourse` in second normalform (with a composite primary key consisting of `StudtenNr` and `Course`)

`StudentNr`	`Course`
1022	101-07
1022	143-01
1022	159-02
4123	201-01
4123	211-02
4123	214-01

The rules for the third normalform (3NF) require that there is no functional dependency on non-key columns. In our example, the column `MentorRoom` is functional dependent on column `Mentor` (which is not the primary key), but not on `StudentNr` (which is the primary key). A mentor keeps using the same room, independent from the mentee student. Therefore, we have to

split now into three tables, carving out columns `Mentor` and `MentorRoom` into a separate table (with `Mentor` as the primary key). The second table contains `StudentNr` (primary key) and `Mentor` (foreign key to the newly created table). And finally, the third, unchanged table, contains `StudentNr` (foreign key) and `Course` (which both form the primary key of this table).

Table 1-6. Table `Student` in third normalform (with primary key `StudentNr`)

`StudentNr`	**Mentor**
1022	Jones
4123	Smith

Table 1-7. Table `Mentor` in second normalform (with primary key `Mentor`)

`Mentor`	**MentorRoom**
Jones	412
Smith	216

Table 1-8. Table `StudentCourse` was already in third normalform as well (with a composite primary key consisting of `StudentNr` and `Course`)

se)

StudentNr	Course
1022	101-07
1022	143-01
1022	159-02
4123	201-01
4123	211-02
4123	214-01

This final version is free of any redundancy. Every single piece of information is only stored once. The data model is though rather complex. This complexity comes with a price: It is hard to understand. It is hard to query (because of many necessary joins). And queries might be slow (because of many necessary joins). Therefore, I will introduce you to *Dimensional Modeling* in the next section.

Dimensional Modeling

Data models in third normalform (= fully normalized) avoid any redundancy, which makes them perfect for storing information for applications. Data maintained by applications, can rapidly change. Normalization guarantees, that a change has only to happen in one single place (= content of one single column in one single row in on single table).

Unfortunately, normalized data models are hard to understand. If you look on the ERD of a model for an even simple application, you will be easily overwhelmed by the number of tables and relationships between them. It's not rare that the printout will cover the whole wall of an office and that application developers who use this data model are only confident about a certain part of the data model.

Such data models are also hard to query. As in the process of normalizing multiple tables get created, querying the information in a normalized data models requires to join multiple tables together. Joining tables is expensive. It requires a lengthy query to be written (the lengthier, the higher the chance for making mistakes; if you don't believe me, re-read the chapters about "Joins" and "Join Path Problems") and it requires to physically join the outspread information from different tables by the database management system. The more joins, the slower the query.

Therefore, let's introduce *Dimensional Modeling*. You can look at this approach as a (very good) compromise between a single table and a fully normalized data model. Dimensional models are sometimes referred to as

denormalized models. As less as I like the term *normalized*, as much do I dislike the term *denormalized*. *Denormalizing* could be easily misunderstood as the process to fully reverse all steps done during *normalizing*. That's wrong. A *Dimensional Model* reintroduces some redundancy but does not undo all the efforts of bringing a data model into third normal form.

Remember, the ultimate goal is to create a model, which is easy to understand and use (by the report creators) and which allows for fast query performance. It is very common for data warehouses (DWH), OLAP systems (Online Analytical Processing), also called cubes, and is the optimal model for Power BI and Analysis Services Tabular.

For designing a dimensional model, it is crucial to identify an attribute (or table) either as a *dimension* (hence the name *Dimensional Modeling*) or as a *fact*:

> *Dimension*
> A dimension table contains answers to questions like: How? What? When? Where? Who? Why? Those answers are used to filter and group information in a report. This kind of table can be wide (= it can contain loads of columns). Compared to facts, dimension tables are relatively small in terms of the number of rows ("short"). Dimension tables are on the "one" side of a relationship. They have a mandatory primary key (so they can be referenced by a fact table) and contain columns of all sorts of data types. In a pure *Star Schema*, dimension tables do not contain foreign

keys.

Fact

A fact table tracks real world events, sometimes called transactions, details, or measurements. It is the core of a data model, and its content is used for counting and aggregating in a report. You should pay attention, that you keep a fact table narrow (only add columns if really necessary), as compared to dimensions, fact tables are relatively big in terms of the number of rows. Fact tables are on the “many” side of a relationship. If there is not a special reason, then a fact table will not contain a primary key, because a fact table is not – and never should be – referred by another table and every bit you save in each row, sums up to a lot of space, when multiplied by the number of rows. Typically, you will find foreign keys and (mostly) numeric columns. The latter can be of additive, semi-additive or non-additive natures. Some fact tables contain transactional data, others snapshots or aggregated information.

Depending on how much you denormalize the dimension tables, you will end up with a *Star Schema* or a *Snowflake Schema*. In a *Star Schema* dimensional tables do not contain foreign keys. All relevant information is already stored in the table in a fully denormalized fashion. That’s the preferred way for a dimensional model. Only if you have certain reasons, you might keep a dimension table (partly) normalized and split information over more than one table. Then some of the dimension tables contain a foreign key. *Star Schema* is preferred over a *Snowflake Schema,* because in comparison a *Snowflake*

Schema

- has more tables (due to normalization)
- takes longer to load (because of the bigger amount of tables)
- makes filter slower (due to necessary additional joins)
- makes the model less intuitive (instead of having all information for a single entity in a single table)
- impede the creation of hierarchies (in Power BI / Analysis Services Tabular)

Of course, a dimension contains redundant data, due to denormalizing. In a data warehouse scenario this is not a big issue, as there are not several processes who add and change rows to the dimension table, but only one single one (as explained below in section “Extract, Transform, Load”).

The number of rows and columns for a fact table will be given by the level of granularity of the information you want or need to store within the data model. It will also give the number of rows of your dimension tables. The next section talks about this important part.

Granularity

Granularity means the level of detail of a table. On the one hand, we can define the level of detail of a fact table by the foreign keys it contains. A fact table could track sales per day, or it could track sales per day and product, or

by day, product and customer. This would be three different levels of granularity.

On the other hand, we can also look on the granularity in the following terms:

Transactional fact
The level of granularity is an event. All the details of the event are stored (not aggregated values).

Aggregated fact
In an aggregated fact table some foreign keys are left out and the rows are grouped and aggregated on the remaining foreign keys. This can make sense when you want to save storage space and/or make queries faster. An aggregated fact table can be part of a data model additionally to the transactional fact table, when the storage space is not so important, but query performance is. In the chapters about performance tuning you will learn more about how to improve query time with the help of aggregation tables.

Periodic snapshot fact
When you do not reduce the number of foreign keys, but reduce the granularity of the foreign key on the date table, than you have created a periodic snapshot fact table. For example, you keep the foreign key to the date table, but instead of storing events for every day (or multiple events per day) you reference only the (first day of the) month.

Accumulated snapshot fact

In an accumulated snapshot table aggregations are done for a whole process. Instead of storing a row for every step of a process (and storing e. g. the duration of this process) you store only one single row, covering all steps of a process (and aggregating all related measures, like the duration).

No matter, which kind of granularity you choose, it’s important that the granularity of a table stays constant for all rows of a table. For example, you should not store aggregated sales per day in a table, which is already on the granularity of day and product. Instead, you would create two separate fact tables. One with the granularity of only the day, and a second one with granularity of day and product. It would be complicated to query a table, in which some rows are on transactional level, but other rows are aggregated. This would make the life of the report creator hard, and not easy.

Keep also in mind, that the granularity of a fact table and the referenced dimension table must match. If you store information by month in a fact table, it is advised to have a dimension table with the month as the primary key.

Now that we know how the data model should look like, it is time to talk about how you can get the information of your data source into the right shape. The process is called “Extract, Transform and Load” and I introduce it in the next section. In later chapters I will show you concrete tips, tricks and scripts of how to use Power BI, DAX, Power Query and SQL to implement

transformations.

Extract, Transform, Load

By now I have hopefully made it clear, that a data model which is optimized for an application looks very different from a data model for the same data, which is optimized for analytics. The process of converting the data model from one type to another is called *Extract, Transform and Load* (ETL):

Extract
"Extract" means to get the data out of the data source. Sometimes the data source offers an API, sometimes it is extracted as files, sometimes you can query tables in the application's database.

Transform
Transforming the source data starts with easy tasks as giving tables and columns user friendly names (as nobody wants to see "EK4711" as the name of a column in a report) and covers data cleaning, filtering, enriching, etc. This is where converting the shapes of the tables into a dimensional model happens. In the chapters about "Building a Data Model" you will learn concepts and techniques to achieve this.

Load
As the source system might not be available 24/7 for analytical queries (or ready for such queries at all) and transformation can be complex as well, it

is recommended to store the extracted and transformed data in a way, where it can be queried easily and fast, e. g. in a Power BI dataset in the Power BI Service or in an Analysis Services database. Storing it in a relational data warehouse (before making it available to Power BI or Analysis Services) makes sense in most enterprise environments.

The ETL process is sometimes also described as the kitchen of a restaurant. The cooks have dedicated tools to process the food and put in all their knowledge and skills to make the food both, good-looking and tasteful, when served on a plate to the restaurant’s customer. This is a great analogy to what happens during ETL: We use tools and all our knowledge and skills to transform raw data into savory data which makes appetite for insights (hence the name of the author’s company). Such data can then easily be consumed to create reports and dashboards.

As the challenge of extracting, transforming, and loading the data from one system to another is widespread, there are plenty of tools available. Common tools in Microsoft’s Data Platform family are SQL Server Integration Services, Azure Data Factory, Power Query, and Power BI dataflows. You should have one single ETL job (e. g. one SQL Server Integration Services package, one Azure Data Factory pipeline, one Power Query query or one Power BI dataflow) per entity in your data warehouse. Then it is straighforward to adopt the job in case the table changes.

Sometimes people refer not to ETL, but to ELT or ELTLT, as the data might

be first loaded into a staging area and then transformed. I personally don’t think it so important if you first load the data and then transform it, or the other way around. This is mostly given as a fact by which tool you are using (if you need or should first persist data before you transform it, or if you can transform it “on-the-fly” when loading the data). The only importance is that the final result of the whole process must be accessible easily and fast by the report users, to make their life easier (as postulated in the introduction to this chapter).

Implementing all transformations, before users query the data is crucial. And I think it is crucial as well to apply transformations as early as possible. If you possess a data warehouse, then implement the transformations there (via SQL Server Integration Services, Azure Data Factory, or simply views). If you don’t have (access to) a data warehouse, then implement the transformations in Power BI data flow or Power Query in a Power BI dataset. Only implement the transformations in the report layer as the last resort (better to implement it there instead of not implementing it at all). The reason for this rule is, that, the “earlier” in your architecture you implement the transformation, the more tools and users can use them. Something implemented in the report only, is only available to the users of the report. If you need the same logic in an additional report, you need to re-create the transformation in the additional report (and face all consequences of code-duplication, like higher maintenance effort for code-changes and the risk of different implementations of the same transformation, leading to different results). If you do the transformation in Power Query (in Power BI or in

Analysis Services), then only users and tools with access to the Power BI dataset or Analysis Services Tabular database benefit from them. While, when you already implement everything in the data warehouse layer (which might be a relational database but could be a data lake or delta lake as well or anything else which can hold all the necessary data and allows for your transformations), then a more widespread population of your organization have access to clean and transformed information, without repeating transformation again (and you connect Power BI to those tables and don’t need to apply any transformations).

Every concept and idea I introduced so far is based on the great work of two giants of data warehousing: Ralph Kimball and Bill Inmon. It is time that I introduce you to them.

Ralph Kimball and Bill Inmon

A book about data modelling would not be complete, without mentioning (and referencing to) Ralph Kimball and Bill Inmon. Both are the godfathers of data warehousing. They invented many concepts and solutions for different problems you will face when creating an analytical database. Their approaches have some things in common but show also huge differences. On their differences they never found compromises and they “fought” about them (and against each other) in their articles and books.

For both, dimensional modelling (facts and dimensions) play an important

role as the access layer for the users and tools. Both call this layer a *Data Mart*. But they describe the workflow and the architecture to achieve this quite differently.

For Ralph Kimball the data mart comes first. A data mart contains only what is needed for a certain problem, project, workflow, etc. A data warehouse does not exist on its own but is just the collection of all available data marts in your organization. Even when “agile project management” was not (yet) a thing, when Ralph Kimball described his concept, they clearly match easily. Concentrating in smaller problems and creating data marts for them allows for quick wins. Of course, there is a risk that you do not always keep the big picture in mind and end up with a less consistent data warehouse, as dimensions are not as conformed as they should be over the different data marts. The *Enterprise Data Bus*’s task is to make all dimensions conformed. Ralph Kimball retired in 2015, but you find useful information at *https://www.kimballgroup.com/* and his books are still worth a read. Their references and examples to SQL are still valid. He didn’t mention Power BI or Analysis Services Tabular, as this was only emerging then.

On the opposite, Bill Inmon favors a top-down approach: You need to create a consistent data warehouse in first place. This central database is called the *Corporate Information Factory* and it is fully normalized. Data marts are then derived from the *Corporate Information Factory* where needed (by denormalizing the dimensions). While this will guarantee a consistent database and data model, it surely will lead to a longer project duration while

you collect all requirements and implement them in a then consistent fashion. His ideas are collected in “Building a Data Warehouse” (2005, Wiley) and are worth read as well. Bill Inmon also supports the Data Vault modeling approach (“Data Vault & Other Anti-Patterns”) and is an active publisher of books around data lake architecture.

Over the years many different data modelling concepts have been developed. And many different tools to build reports and support ad-hoc analysis have been created. In the next section I will describe them as anti-patterns. Not because they are bad in general, but because Power BI and Analysis Services Tabular are optimized for the star-schema.

Data Vault & Other Anti-Patterns

I will not go into many details of how you can implement a *Data Vault* architecture. Its though important to lay out, that a Data Vault is merely a data modeling approach which makes your ETL flexible and robust against changes in the structure of the data source. Data Vault’s philosophy is to postpone cleaning of data to the business layer. As easy this approach makes the live of the data warehouse / ETL developers, as hard it will make the life of the business users. But remember: The idea of this book is to describe how you can create a data model which makes the end-users life easier.

A Data Vault model is somewhere between third normal form and a star schema. Proponents of the Data Vault claim rightfully, that such a data model

can also be loaded into Power BI or Analysis Service Tabular. The problem is though: You can load *any* data model into Power BI and Analysis Services Tabular – but you will pay a price when it comes to query performance (this happened to me with the first data model I implemented with Power BI; even when the tables contained just a few hundred rows, the reports I built where really slow). You will sooner or later suffer from overcomplex DAX calculation, too.

That’s why I try to strongly convince you, not using any of the following data model approaches for Power BI and Analysis Services Tabular:

Single table
I already described my reasoning in “A Single Table To Store It All”.

A table for every source file
This is a trap non-IT users easily step into. A table should contain attributes of one single entity only. Often, a flat file or an Excel spreadsheet contains a report and not information limited to one single entity. Chance are high, that when you create a data model with a table per file, the same information is spread out over different tables and many of your relationships show a many-to-many cardinality due to a lack of primary keys. Applying filters on those attributes and writing more than just simple calculations can quickly start to be a nightmare. Sometimes this “model” is referred to as *OBT* (one big table).

Fully normalized schema

Such a schema is optimized for writing, not optimized for querying. The number of tables and necessary joins makes it hard to use and leads to slow query response time. Chances are high that query performance is less than optimal and that you will suffer from "Join Path Problems".

Header – Detail

Separating e. g. the order information and the order line information into two tables requires to join two relatively big tables (as you will have loads of orders – and loads of order lines, representing the different goods, per order). This additional join will make queries slow and DAX more complex than necessary, compared to combining the header and detail table into one single fact table. The joined table will contain as many rows as the detail table already has and as many columns as the two tables combined, except for the join key column, but will save the database management system from executing joins over two big tables.

Key-Value

A Key-Value table is a table with basically just two columns: a key column (containing e. g. the string "Sales") and a value column (containing e. g. "100"). Such a table is very flexible to maintain (for new information you just add a new row with a new key, e. g. "Quantity"), but it is very hard to query. In chapter 3 I write in length about the challenges key-value-pair tables bring, and how to overcome them in order to transform them into a meaningful table.

The reason why I describe these as anti-patterns is not that these modelling approaches, in an objective point of view, per se are worse than star schema. The only reason is, that many reporting tools benefit from a star schema so much, that it is worth to transform your data model into one. The only exceptions are tools like Power BI Paginated Reports, which benefit from (physical or virtual) single tables containing all the necessary information.

The *VertiPaq engine* (which is the storage engine behind Power BI, Analysis Services Tabular, Excel's PowerPivot and SQL Server's Column Store index) is fully optimized for star schema with every single fiber. You should not ignore this fact.

While you can write a letter in Excel and do some simple calculations in a table in a Word document, there are good reasons why you would write a letter with Word and create the table and its calculations in Excel. You would not start complaining how hard it is to write a letter in Excel, or that are many features to do your table calculations are missing in Word. Your mindset towards Power BI should by similar: You can use any data model in Power BI, but you should not start complaining about the product unless you have your data model as star schema.

Key Takeaways

Congratulations on finishing the first chapter of this book. I am convinced

that all the described concepts are crucial for your understanding of data models in general, and for all the transformations and advanced concepts I will talk about in the rest of the book. Here is a short refresher of what you learned so far:

- You learned about the basic parts of a data model: tables, columns, relationships, primary keys, and foreign keys.
- We talked about different ways of combining tables with the help of set operators and joins, and which kind of problems you can face when joining tables.
- Normalized data models are optimized for write operations, that's why they are the preferred data model for application databases. Dimensional modelling re-introduces some redundancy to make them easier to understand and to allow for faster queries (as there are less joins necessary).
- Transforming of the data model (and much more) is done during the ETL, which extracts, transforms and loads from data sources into the data warehouse.
- I gave you a rough overview about the contrary ideas of the two godfathers of data warehouses, Ralph Kimball and Bill Inmon.
- At the end I pointed out, why it is so important to stick to a star schema, when it comes to Power BI and Analysis Services Tabular. Other approaches have their value – but they or not optimal for the *VertiPaq engine*, which handles all the data queried in Power BI, Analysis Services Tabular, Excel's PowerPivot and SQL Server's Column Store index.

Chapter 2. Building a Data Model

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sevans@oreilly.com*.

Traditionally we speak of OLTP (**O**n*l*ine *T*ransactional *P*rocessing) databases on the one hand and OLAP (*O*n*l*ine *A*nalytical *P*rocessing) databases on the other hand. The term "online" is not related to the internet here, but means, that you query a database directly instead of triggering and waiting for an asynchronous batch job, which runs in the background – something you might only have seen in your career when you are about my age (or even older). "Transactional" means that the purpose of the database is to store real-world events (transactions). This is typical for databases behind any application your mind can come up with: the software

your bank uses to track the movement of money or the retailer which keeps track of your orders and their delivery. Databases for such use cases should avoid redundancy under all circumstances: A change of your name should not end up in a complicated query to persist the new name through several tables in the database, but only in one single place.

This book concentrates on analytical queries in general and on Power BI and Analysis Services in particular. Therefore, when I speak of a data model in this and all following chapters, I mean data models built for analytical purposes, OLAP databases. For Power BI and Analysis Services the optimal shape of the data model is the dimensional model. Such databases hold the data for the sole purpose of making analytical queries and reports easy, convenient and fast (*Make the report creators life easier*.).

Building an analytical database (and transforming data from data sources which were built with other goals in mind into a dimensional model) is mostly not easy and can be a challenge. This chapter will help you to understand those challenges and how to overcome them. As you already learned in Chapter 1, you need to normalize the fact tables and denormalize the dimension tables, which is what I start describing in the next section. You should also add calculations, transform flags and indicators into meaningful information to make the data model ready to use for reports. I recommend that you build a dedicated date (and maybe an additional time) dimension so that the report creator does not need to fumble with a date column and extract the year, month etc. for filters and groupings. Some dimensions may play

more than one role within a data model, and you will learn how to model such cases. We will discuss the concepts of slowly changing dimensions and how to bring hierarchies into the right shape, so they can be used in Power BI.

Remember from Chapter 1: Normalizing and denormalizing are terms to describe to remove or to add redundancy to a data model. A database for the purpose of storing information for an application should be fully normalized. Analytical databases, on the other side, should contain redundant information, where appropriate. In the next two sections you will learn where redundancy should be avoided in an analytical database as well and where you should explicitly make information redundant.

Normalizing

Normalizing means to apply rules to the data model with the ultimate goal to avoid redundancy. In Chapter 1 you learned about the importance of normalizing a data model and why this is so important for OLTP (Online Transactional Processing) databases.

Normalizing is also necessary for fact tables in a Dimensional Model. As you learned in Chapter 1, fact tables are the biggest tables in a data warehouse in terms of number of rows, and they constantly get more rows added. Every bit and byte we can save within a single row by optimizing the amount and type of columns we have, is more than welcomed. Think about the following: If

you save a single byte per row in a table containing one million rows, you save one megabyte of data. If you save ten bytes in a table containing one billion rows, you save 10 gigabytes of data for this table. Less data will put less pressure on the given infrastructure. And scanning less data will also lead to faster reports.

Only when you are able to identify a table as a fact table you can fully normalize them, just as described in Chapter 1. Typically, if your data source is a flat file (e. g. an Excel spreadsheet someone created or extracted from a database system) chances are high, that a model created as one table per Excel worksheet will be too denormalized, hence the worksheets need to be normalized. The extreme case is that of a data model consisting of one big table (OBT), where all the information resides in one single table. You should avoid this, as you do not want to have any tables in the model, which are long (= many rows) and wide (= many columns) simultaneously.

You will also face situations where details for an entity are spread over different sources, tables or files. That's where you need to denormalize.

Denormalizing

Denormalizing means that you intentionally introduce redundancy into a table: The same piece of information is repeated over several rows within a table, because several rows share the same information (e. g. as several

customers reside in the same country or several products are part of the same product category). This happens every time you add a column to a table, which contains information not unique per primary key of the table.

When you model a natural hierarchy within one single table, you will face redundancy. For example, one product category can consist of more than one single product. If you store the product name together with the category name in a single table, the name of a category will appear in several rows.

Or think of a table containing a row for each day of a year with a column containing the date. Adding a column for the year or the month will introduce redundancy as a given year or a given month will appear in several rows. On top of that, storing the year and month additionally to the date is redundant from the point of view that the year and the month can always be calculated by applying a function on the date column. In an OLTP database such a redundancy is unwanted and should be avoided under all circumstances. In an analytical database this type of redundancy is wanted and recommended, for several reasons:

- Having all information about an entity in one single place (table) is user friendly. Alternatively, e. g. product related information would be spread out over several tables like, `Product`, `Product Subcategory`, and `Product Category`.
- Additionally, having all information pre-calculated at hand as needed is more user-friendly (instead of putting the burden onto the report-user and

the used report-tool to calculate the year from a date, for example).

- Joining information over several tables is expensive (in terms of query performance and pressure on the resources). Reducing the number of joins to satisfy a query will improve the performance of the report.
- The relatively small size of dimensions allows for added columns without a huge impact onto overall size of the model. (In the case of Power BI and Analysis Services this problem is even smaller, as the storage engine's automatic compression algorithm is optimized for such scenarios). Therefore, the backdraft of denormalizing is not as huge of a problem when it comes to storage space, as you might think.
- Power BI Desktop and Analysis Services Tabular are optimized for a fully denormalized star-schema.

Long story, short: All dimension tables should be fully denormalized, to form a star-schema.

Furthermore, you should enrich the source's data by adding all sorts of calculations (again, to remove the burden of creating these from the report-user). That's what I will discuss in the next section.

Calculations

It's a good idea to add a calculation as early as possible in your stream of data. Keep in mind though, that only additive calculations can be (pre-

)calculated in the data source. Semi- and non-additive calculations must be calculated as measures (in case of Power BI and Analysis Services this means: in the DAX language):

Additive

Many calculations can be calculated on top of results of lower granularity. The given quantity sold in a day can be added up to monthly and yearly results. The sales amount (calculated as the quantity multiplied by the appropriate price) can be added over several products.

Semi-additive

The result of a semi-additive calculation can be aggregated over all dimensions, except the date dimension. A typical example is stock levels. Stock levels are stored as the number of products available on a certain day. If you look at a certain day you can add the stock levels over several warehouses for a product: You can safely say that we have 5 kg of vanilla ice cream if there is 3kg in one freezer and another 2kg in a second freezer. But it does not make sense to add up the individual stock level of different days: When we had 5kg yesterday and today only 1kg is left, then adding these two numbers up to 6kg gives a meaningless number. Thus, the calculation formula needs to make sure, to use only the data from the most current day within the selected timeframe.

Non-additive

Some calculations cannot be aggregated at all. This covers distinct counts

and all calculations containing a division operator in its formula (e. g. average, percentage, ratio). Adding up results of such a calculation does not make any sense: Instead of aggregating the results, the formula must be executed upon the aggregated data: counting the distinct customers over all days of the month (instead of adding up the number of distinct customers per day) or dividing the sum of margin by the sum of sales amount (instead of dividing the margin by the sales amount of each individual sales row and then summing up those results).

Formulas can also be applied to non-numeric values. In the next chapter you will find out why and how this should be done.

Flags and Indicators

In most cases, reports showing rows of “Yes” and “No” values or abbreviations like “S” or “M” are hard to read. Therefore, all flags and indicators delivered by the source system must be converted into meaningful text. For example:

- FinishedGoodFlag with content 0 or 1 should be transformed accordingly into text “no salable” or “salable”.
- Productlines “R”, “M”, “T”, or “S” should be transformed accordingly into text “Road”, “Mountain”, “Touring”, or “Standard”.
- Column Class with values “H”, “M”, or “L” should be transformed

accordingly into “High”, “Medium”, or “Low”.

- Styles containing “W”, “M”, or “U” should be transformed accordingly into “Women’s”, “Men’s”, or “Unisex”.
- In general blank (or null) should be avoided, but replaced by a meaningful text: “unknown”, “N/A”, “other”, etc. Depending on the context, a blank value could be transformed to different texts (to distinguish an “unknown” value from “other”) within the same column.

Do you create reports on data which is not related to any point in time? Writing this book, I thought hard about it and could not remember a single report I created which did not either filter or aggregate on dates, or even both. Of course, this does not mean that such report does not exist. But it makes me confident that such reports are not so common. Therefore, you should prepare your data model to make handling date and time easy for the end-user. The next section is exactly about this.

Time and Date

It’s very rare to build a data model upon data, which does not bear any relation to a point in time. Therefore, a date dimension is very common in the majority of data models. The idea of a date dimension is two-folded:

- Create columns for all variants of a date information which will be later used in the reports. Year, month number, month name, name of the

weekday, week number, etc. are common examples. The report tool shall not cover this, but just show the pre-calculated columns. Therefore, add a column for every variation needed in the report (e. g. “December 2023”, “2023-12”, “Dec”, …).

- Having a table with one row per day of a calendar year. This allows you to calculate a duration in days and is mandatory if you want to use the built-in time intelligence functions in DAX (which we will cover in Chapter 10).

Week numbers can be tricky, by the way. There are basically two definitions – which only deviate from each other in certain years. If you do not pay attention to the right calculation of the calendar week, you might end up with a sudden surprise in one year. Wikipedia got you covered in case you need to find out which definition is the one the report users expect (*https://en.wikipedia.org/wiki/ISO_week_date*).

A time dimension (having rows for hours and minutes within a day) on the other hand is in fact very rare in my experience. It’s important that you separate the time dimension from the date dimension, so both can be filtered independently from each other. Furthermore, splitting a timestamp into a date and a time portion, minimizes the number of distinct rows: To cover a full calendar year, you need 365 (or 366 for leap years) rows in the date dimension, and 1440 (= 24 hours multiplied by 60 minutes) rows for a time dimension to cover every minute. For every new year, you add another 365 (or 366) rows in the date table. If you stored this information together in one

single datetime table, you would end up with 525600 (365 days times 24 hours times 60 minutes) rows. For every year, you would add another 525600 rows in the datetime table.

Talk to your end-users to find out, on which granularity level of the time they need to filter and group information. If the lowest granularity is e. g. only by hour, make sure to round (or trim) the timestamp in your fact table to the hour and create a time dimension with only 24 rows.

Role-Playing Dimensions

Sometimes one single entity can play different roles in a data model. A person could simultaneously be an employee and a customer as well. A year could have the meaning of an order year and/or the shipping year. These are examples of role-playing dimensions.

Assigning different roles can be achieved with the following approaches:

- Load the table only once, and then assign different roles by creating several relationships between the dimension table and the fact table, according to its roles. For example, you create two filter relations between the `Date` dimension and the `Sales` fact table. One where you connect the `Date`'s date column first with the `Sales`' order date and second with the `Sales`' ship date. The report creator needs then a way to specify, which role the dimension should play in a visualisation.

- Load the table twice into the data model, under two different names. For example, you would load the `Date` table first as `Order Date` and second as `Ship Date` table. (Make sure, that the column names are unique throughout the data model, by e. g. adding the `Order` or `Ship` prefix to the column names as well: `Year` becomes `Order Year` etc.) You would then create filter relationships between the `Sales` fact and those two tables. The report creator chooses either the `Order Date` or the `Ship Date` table according to the needs.

Slowly Changing Dimensions

The value for a column of a row in a dimension table is usually not constant but can change over time. The question we need to clarify with the business users is if it is important to keep track of changes – or if we can just overwrite the old information with the new information. A decision needs to be made per column of a dimension table (maybe the business wants to overwrite any changes of the customer’s name but keep a historic track of the changes of the customer’s address).

We talk about slowly changing dimensions when chances of the attributes are happening only once in a while. If the information for a dimension changes often (e.g. every day) you might capture the changes of this attribute not in the dimension table, but in a fact table instead. Unfortunately, there is not a clear line here on how to distinguish slowly changing dimensions from rapid

changing dimensions.

While Ralph Kimball was very creative with creating new terms for the challenges of analytical databases, he came up with a rather boring way of naming the different types of slowly changing dimensions, he just numbered them:

Type 0: Retain Original

Usually only a small set of dimensions (and their columns) shall never change. For example, August 1 2023 will always be a Tuesday. And will always be part of month August and year 2023. This will never change – it's not necessary to implement a way of updating this information.

Type 1: Overwrite

When the name of a customer changes, we want to make sure to correct it and display the new name in all reports – even in reports for that past (where an old version of the report may show the old name; re-creating the same report now will show the new name). Maybe we want to store some additional columns in the table, like, when the change happened and who (or which ETL process) did the change. In Table 2-1, you see a table containing three rows, enriched with a `ChangedAt` and a `DeletedAt` column, which represent the day of modification (or creation) and invalidation, respectively.

Table 2-1. SCD Type 1 before the change

AlternateKey	Region	ChangedAt	DeletedAt
0	NA	2023-01-01	
1	Northwest	2023-01-01	
10	United Kingdom	2023-01-01	

Let’s assume, that we get the new data, as laid out in Table 2-2: the row for region “NA” was removed from the data source, the name of region “Northwest” was translated to German language “Nordwest”, the row for “United Kingdom” stayed unchanged and a new row for “Austria” was added.

Table 2-2. SCD Type 1 changed rows

AlternateKey	Region
1	Nordwest
10	United Kingdom
11	Austria

As you can see in Table 2-3, in a Type 1 solution, the row for “NA” will not be removed, but marked as deleted by setting the `DeletedAt` column to the timestamp of removal. The row for “Northwest” will be changed to “Nordwest” and the `ChangedAt` timestamp will be updated. “United Kingdom” will stay unchanged. And the new row for “Austria” is added, with a `ChangedAt` set to the current day.

Table 2-3. SCD Type 1 after the changes

AlternateKey	Region	ChangedAt	DeletedAt
0	NA	2023-01-01	**2023-08-15**
1	**Nordwest**	**2023-08-15**	
10	United Kingdom	2023-01-01	
11	Austria	**2023-08-15**	

This is a very common type of slowly changing dimension.

Type 2: Add New Row

If you want to update a column, but need to guarantee, that a report made for the past does not reflect the change (but stays the same, even if created

today), then we need to store two versions. One version, reflecting the status before the change, and a new version, reflecting the status after the change. An example could be the address (and region) of a customer. If the customer moves, maybe we only want to assign sales made after the customer moved to the new region but want to keep all previous sales in the old region.

Slowly Changing Dimension Type 2 achieves this by creating a new row in the dimension table for every change we want to keep track. It is important to mention that for this solution we need to have a surrogate key as the primary key in place, as the business key will not be unique after the first change. Customer "John Dow" will have two rows in the customer table. One row before the change, one row after the change (and several further rows after more changes happened). All sales before the change use the old versions surrogate key as the foreign key. All sales after the change use the new versions surrogate key as the foreign key. Querying is therefore not big of an issue (as long as the report users do not need to select a certain version of the customer to be used for a report for any point in time; chapter 9 will have you covered to implement this request). In Table 2-4 you see a table which contains the additional columns to describe the timespan, when the row is/was valid (`ValidFrom` and `ValidUntil`). This looks somehow similar to the Type 1 solution. In the example I kept **ValidUntil** empty for rows without a invalidation. Alternatively, you could also use a timestamp far in the future (e. g. December 31 9999).

Table 2-4. SCD Type 2 before the change

AlternateKey	Region	ValidFrom	ValidUntil
0	NA	2023-01-01	
1	Northwest	2023-01-01	
10	United Kingdom	2023-01-01	

Let’s assume, that we get the same new data, as laid out in Table 2-5: the row for region “NA” was removed from the data source, the name of region “Northwest” was translated to German language “Nordwest”, the row for “United Kingdom” stayed unchanged and a new row for “Austria” was added.

Table 2-5. SCD Type 2 changed rows

AlternateKey	Region
1	Nordwest
10	United Kingdom

11	Austria

As you can see in Table 2-6, in a Type 2 solution as well, the row for “NA” will not be removed, but marked as deleted by setting the `ValidUntilAt` column to the timestamp of removal. For the row, containing “Northwest” the `ValidUntil` timestamp will be updated and a new version for the same `AlternateKey`, but region “Nordwest” will be inserted into the row. “United Kingdom” will stay unchanged. And the new row for “Austria” is added, with a `ValidFrom` set to the current day.

Table 2-6. SCD Type 1 after the changes

AlternateKey	Region	ValidFrom	ValidUntil
0	NA	2023-01-01	**2023-08-15**
1	Northwest	2023-01-01	**2023-08-15**
10	United Kingdom	2023-01-01	
1	Nordwest	**2023-08-15**	
11	Austria	**2023-08-15**	

WARNING

You need to keep an eye on how many changes are to be expected for the dimension on average in a certain period of time, as this approach will let the dimension table grow in terms of rows.

This is a very common type of slowly changing dimension as well.

Type 3: Add New Attribute

Instead of creating a new row for every change, Type 3 keeps a dedicated column per version. Obviously, you need to decide upfront of how many versions you want to keep track of, as you need to provide one column per version.

New versions will therefore not let the table grow, but the number of versions you can keep per entity is limited. Querying can be a bit of an issue, as you need to query different columns, depending on if you want to display the most current value of an attribute or one of the previous versions.

I have never implemented this type of slowly changing dimension for one of my customers. But it may still be a useful approach for your use case.

Type 4: Add Mini-Dimension

This approach keeps tracks of the changes in new rows, but in a separate table. The original table shows the most current version (as Type 1 does) and the older versions are archived in a separate table (which can hold as

many older versions as you need). Querying the most current version is easy. Showing older versions involves a more complex query for joining a fact table to the correct row of the archive table. Or you would store both the foreign key to the original table and a second foreign key to the matching rows in the mini-dimension. New versions do not change the number of rows in the original table but will certainly do in the extra table.

Again, I have never implemented this type of slowly changing dimension for one of my customers. But it may still be a useful approach for your use case.

NOTE

The rest of the types are more or less combinations of the previous versions. I am sure they have their use cases – but I never had to implement them, as Type 1 and Type 2 were sufficient for my client's needs so far, that's why I just give you a short overview here instead of a in-depth description:

- Type 5: Add Mini-Dimension and Type 1 Outrigger
- Type 6: Add Type 1 Attributes to Type 2 Dimension
- Type 7: Dual Type 1 and Type 2 Dimensions

At *https://www.kimballgroup.com/2013/02/design-tip-152-slowly-changing-dimension-types-0-4-5-6-7/* you find more about these types.

Hierarchies

Hierarchical relationships can be found in real-life in many situations:

- Product categories (and their main- and subcategories)
- Geographical information (like continent, regions, countries, districts, cities, etc.)
- Time and date (like year, month, day, hour, minute, second, etc.)
- Organigram (every employee reports to another employee with the CEO at the top)

I am convinced that you will have some hierarchical structures in your data model(s) as well. From a technical perspective, the latter example (organigram) is different from the other examples. Typically, you store year, month, day, etc. in separate columns of a dimension table to represent the hierarchy. This doesn't necessarily apply to organigrams, which are a so-called parent-child-hierarchy. There are plenty of ways of storing the parent-child relationships in a table. One way is that an employee references another employee (which is a different row within the same table) over a simple foreign key relationship. This is called a self-referencing relationship because the `Employee` table contains both the primary key and the foreign key used in this relationship. The employee's `Manager ID` references another employee's Employee ID. This is a very efficient way of storing this information, but it's hard to query because you need to somehow travers the organigram from one table to the other.

You can either use a recursive Common Table Expression (CTE) written in SQL to collect information from different levels. Or you could write a recursive function in T-SQL. You can also solve this in DAX (chapter 7) and

Power Query (Chapter 11). Any way, Power BI asks you to to create a so-called materialized path per row in the employees table. Figure 2-1 shows an example, what the materialized path could look like for a bunch of nodes in a hierarchy.

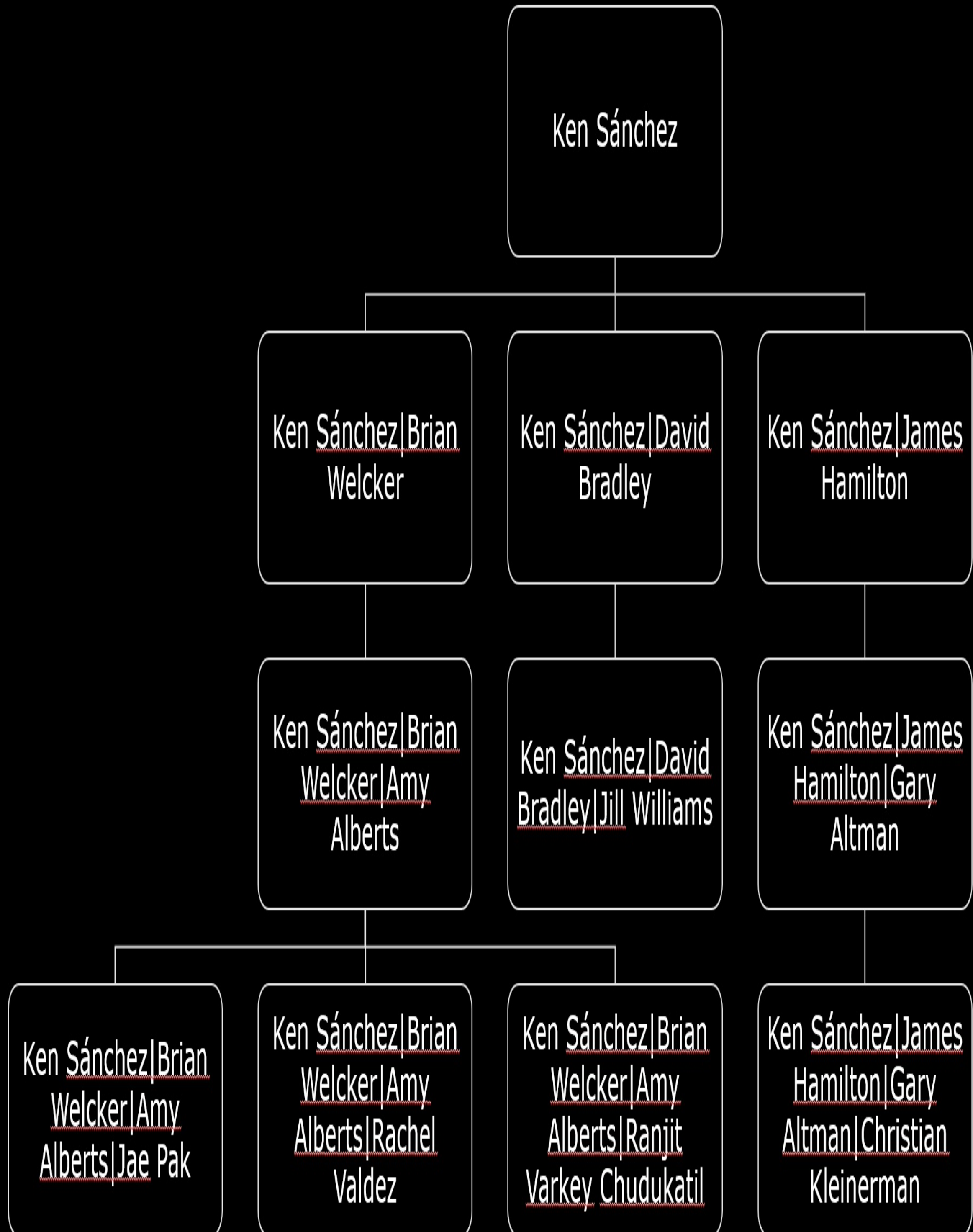

Materialized Path
Ken Sánchez
Ken Sánchez|Brian Welcker
Ken Sánchez|David Bradley
Ken Sánchez|James Hamilton
Ken Sánchez|Brian Welcker|Amy Alberts
Ken Sánchez|David Bradley|Jill Williams
Ken Sánchez|James Hamilton|Gary Altman
Ken Sánchez|Brian Welcker|Amy Alberts|Jae Pak
Ken Sánchez|Brian Welcker|Amy Alberts|Rachel Valdez
Ken Sánchez|Brian Welcker|Amy Alberts|Ranjit Varkey Chudukatil
Ken Sánchez|James Hamilton|Gary Altman|Christian Kleinerman

Figure 2-1. MaterializedPath

The materialized path is simply a string containing a reference to the current node and all its parent nodes. This example uses the names of the employees, concatenated with a pipe (|) as the delimitor. I used the full names for better readabilty, in reality you should use the primary keys of the nodes (e. g. `EmployeeKey`) instead, of course. The delimitor is necessary, otherwise a materialized path of “123” could be interpreted as node 1 and 23 or as node 12 and 3. Make sure that the delimitor will never be used in the actual values.

A materialized path is a rather convenient way to query. This string can be split into separated columns containing e. g. the name of the managers as one column per level. In this flattened table the name of the CEO will then appear in the column representing level one of all employees. Level two contains the next management level, and so on. You can easily count the number of keys in the materialized path (by counting the number of separators and adding one) to know on which level the employee is within the hierarchy. See an example for employee “Amy Alberts” in Table 2-7

Table 2-7. Materialized path and columns per level

EmployeeKey	ParentEmployeeKey	FullName	PathKey
290	277	Amy Alberts	112|277|29

You made it through chapter 2 and now it is time to wrap it up.

Key Takeaways

This chapter guided you through typical tasks when building a data model:

- You need to denormalize your fact tables and fully normalize your dimension tables to form a star schema.
- It's a good idea to push calculations as much as possible upstream. The earlier you create calculations in your overall data flow, the more people and tools can use the calculation, which avoids having calculations defined on many places.
- You should avoid keeping flags and indicators as they are but transform them into meaningful texts instead.
- Time and Date play a crucial role in many data models. Both should be created in the granularity needed in your reports (e. g. a Date dimension with a row for every single day of the year or a Time dimension for every single hour of a day).
- A single dimension may play more than one single role within the data model. You can either create the table several times in your data model (once per role) or create several relationships from the dimension to the fact table and activate the relationship as needed.
- Modelling of Slowly Changing Dimensions is needed when you want to keep track of changes in the attributes of a dimension. The most common

type is the one, where you create a new row for every new version of the entity.

- There are two different types of hierarchies. One, where you have one column per level. And one, where a child references its parent.

Chapter 3. Use Cases

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sevans@oreilly.com*.

In chapters 1 and 2, you learned to understand the basics of a data model and which steps you need to take to build a data model optimized for analytics. This chapter builds upon these steps. Please make sure that your data model is built upon these steps before you dive into the use cases described in this chapter.

WARNING

If you did not proceed as described in chapters 1 and 2, applying the concepts of the current chapter might be full of frustrations – because the best basis for advanced concept is a star schema. There is no shortcut into advanced topics.

In this chapter I will describe five uses cases, which I see at many of my customers – and therefore assume that there is a high likelihood of you facing them as well sooner or later. The list is, of course, a personal selection – every new project comes with its own challenges. *Binning* is the challenge of not showing the actual value, but a category the value falls into, instead. I will then use the case of *Budget* values to introduce a data model, containing more than one single fact table (which still conforms with the rules of a star schema). I am very excited (and proud of) the section *Multi-language model*. It describes the problem of giving the report-user full control over the display language of the report (read: headlines and content) and the solution I came up with. It is similar with the section on *Key-value pair tables*: A data source for a data model of one of my customers contained all information in a single, unpivoted table, for which I describe the problems and (semi-)automatic solutions to pivot the table, which I could not find a pre-defined solution. Finally, the section on *Combining self-service and enterprise business intelligence* will describe the basis for what can be done in Power BI and Analysis Services in a *Composite model*. In the first use case, *Binning*, I will describe different approaches of solving the same problem – all of them forcing you to think a bit out-of-the-box.

Binning

The term “binning” in this context means, that you do not want to show the actual values (like a quantity of four), but to show a category instead (like “medium” or “between one and five”).

Basically, we have the following options to model this requirement, all of them with different disadvantages and advantages, as described:

- Adding a new column to the table containing the value and making sure to fill it with the correct description of the bin as in Table 3-1. This is the simplest of the options, but not recommended, as you would make the fact table (containing the value to be binned) wider. Also, in case the range of the bin or its descriptive text changes, you would need to update the fact table. In some implementations this will not be possible, because due to the size of the fact table and the run-time of the update statement.

  Table 3-1. Adding a column to the fact table

Date	Product	Quantity	Bin
2023-08-01	A	3	Middle
2023-08-01	B	1	Low
2023-08-02	B	4	Middle
2023-08-03	C	5	High

- Creating a lookup table like in Table 3-2, which consists of two columns: One contains a distinct list of all possible values. The second column contains the value to show for the bin. This is identical to the approach with the lookup table to transform a flag or code into a meaningful text, described in the previous chapter.

Table 3-2. Adding a lookup table containing distinct quanties and their bin

Quantity	Bin
1	Low
2	Low
3	Middle
4	Middle
5	High
6	High

You then create an equi-join between the table containing the values to-be-binned and this lookup table. This looks a bit unusual, as we are used to join primary keys and foreign keys and not e. g. a quantity. But this

solution is easy to implement and has a very good performance.
Another advantage is, that such a table is usually easy to create.
Maintaining the table is easy, in principle, as well. The catch is only if somebody needs to maintain the table by hand and typos happen in some of rows (then a value of three would be assigned to “medum” instead of “medium” and would be shown as a category for itself). Or, if the categories get mixed up by accident (that a value of four is “medium”, but a value of five is set to “small”). Usually such a problem is easy to spot and fix manually. Alternatively, you can use a script to create and maintain the table.
A real backdraft is though, that this idea only works, if we can generate a distinct list of all possible values. Yes, you can add some extra lines for outliers (quantities beyond a thousand, maybe), but if we are not talking about pieces, but about pounds or kilograms, then an unknown amount of decimal digits can be involved as well. Rounding (to the nearest whole number or thousand or million) could though help to overcome this problem.

- The other option is to create a table, containing three columns: One, defining the lower value per category, another one to define the upper value per category, and finally the value to show, when a value falls in between the lower and upper range. You can see an example in Table 3-3.

Table 3-3. Adding a lookup table containing ranges of quantities and their bin

Bin	Low (incl.)	High (excl.)

Low		3
Middle	3	5
High	5	

Such a table is even easier to create. It’s less prone to mistakes, but it involves some extra logic, when assigning the actual values to the category.

NOTE

I also want to point out, that I strongly recommend making one range value (e. g. the lower value) inclusive, and the other one (e. g. the upper value) exclusive. That means that a value falls into a category if it is greater or equal the lower bound, but lower than the upper bound. This has the advantage that you can use the exact same number as the upper bound for one category and as the lower bound for the next category. There will be no gaps, as a value is either lower than the upper bound (and therefore falls into this category), or it is greater or equal than the upper bound (which matches the lower bound of the next category) and therefore falls into the next category. Makes sense?

Another challenge I see often in models I build for my customers is that of combining information of different granularity in one single data model. This is e. g. the case when you combine actual values and their budget, as discussed in the next section.

Budget

I called this section “Budget”, but budget is only one of plenty use cases with the exact same problem. The problem I am talking about is the problem of “multi-fact” models. A “multi-fact” model is a data model, containing more than one single fact table. Such models are sometimes called “galaxies” or “universes”, as they contain more than a single “star”. This makes only sense if those stars have at least one single common dimension. If not, I would recommend creating two independent data models instead.

The definitive goal in a star schema is to add new information to the existing tables only, if possible, and not to create a new table for every extra piece of information. The reason is that joining tables is an expensive operation in terms of report/query runtime. With that said, you should first evaluate if the granularity of the new information matches the granularity of an existing table.

Let’s first look at cases, where we can just add the information without further changes.

Maybe you want to add the information about a the product’s category to the reports (and therefore to the data model). If you already have a table which is on the same or a lower granularity than the product category, e. g. a dimension table `Product` which contains information about individual products, you can simply add `Product Category` to this dimension table.

The granularity of the `Product` table will not change, as you can see in Table 3-4

Table 3-4. Product table with main product category

Product Key	Product Name	Product Category
100	A	Group 1
110	B	Group 1
120	C	Group 2
130	C	Group 3

If the new (factual) information, you want to add, is on the same granularity, you can simply add this as a new column. For example, in a table containing `Sales` amounts in EUR you can simply add a new column containing the `Quantity` in pieces. As long as both the amount in EUR and the quantity in pieces are on the same granularity this is no problem. The granularity of a fact table is given by the foreign keys in the table (e. g. date, product, customer, etc), which did not change in the example shown in Table 3-5

Table 3-5. Adding quantity to a fact table

Date	Product	Sales	Quantity

2023-08-01	A	30	3
2023-08-01	B	20	1
2023-08-02	B	120	4
2023-08-03	C	500	5

The more challenging cases are coming now: If the table to start with is on the granularity of product category (e. g. with `Product Category Key` as its primary key as shown in Table 3-6), then adding the product's key would change the granularity of the table. `Product Category Key` would not be the primary key anymore, as it is expected that there are several products (with individual rows) per `Product Category Key`. The cardinality of relationships from (fact) tables to the dimension table would suddenly change from one-to-many to many-to-many: per row in the fact table there will be several rows in the dimension table. This is something you should avoid, as described in chapter 1. Instead you would keep the existing dimension table on its granularity and introduce a new dimension table with the different granularity.

Table 3-6. Product table with main product category

Product Category Key	Product Category

10	Group 1
20	Group 2

Something similar happens if you want to add facts on a “higher” granularity. While we collect actual sales information on the granularity of day, product, customer, etc., values for a budget are typically only available on a coarser level: per month, per product group, not per customer, etc. One solution is to find an algorithm to split the budget value down to the smaller granularity (e. g. dividing the month’s budget over the days of the month). Another solution is to create a fact table of its own for the budget (s. Table 3-7), hence creating a “multi-fact” data model. Then the relationship between the `Budget` fact table and the `Product` dimension table can only be created on `Product Group` level, which has a cardinality of many-to-many (in neither table the `Product Group` is the primary key). In later chapters I will introduce solutions to overcome this problem.

Table 3-7. A budget it typically on a different granularity than the actual values

Month	Product Group	Budget
2023-08	Group 2	20000

2023-08	Group 3	7000
2023-09	Group 2	25000
2023-09	Group 3	8000

No matter what the reason for a many-to-many cardinality is, it is best practice to introduce a table in between to bridge the many-to-many cardinality and creating two one-to-many relationships instead. For example, you create a table consisting of the distinct product groups. The product group's names (or their keys) would be the primary key of this new table. The relationship from this table to the `Budget` table has then a one-to-many relationship. Likewise, the relationship from this table to the `Product` table is a one-to-many relationship, as well.

Now to something completely different: In a global world your users might expect to get the reports shown in a language of their choice. In the next section I describe a data model which allows for such.

Multi-language Model

A requirement for reports/data models to support several languages can be seen on different levels:

Textual content (e. g. product names)

In my opinion, the most robust solution is, to introduce translations of dimensional values as additional columns to the dimension table, as laid out in Table 3-8 New languages can then be introduced by adding rows to the tables – no change to the data model or report is necessary. The challenge is, that the tables' primary key is then not unique anymore, as for e. g. `Dim1 ID` of value 11 we have now several rows in the table (with different content for the description, and an additional `Language ID`). The primary key becomes a composite key (`Dim1 ID` and `Language ID`), which comes with several consequences we will discuss in the other parts of this book.

Table 3-8. A table containing every dimensional entity per language

Language ID	Dim1 ID	Dim1 Desc
EN	11	house
EN	12	chair
DE	11	Hause
DE	12	Stuhl

Visual elements (e. g. report headline)

As I don’t want to create and maintain several versions of the same report (one for each and every languages) I store all the text for visual elements in the database as well (in a table like in Table 3-9). This can be done via a very simple table, containing three columns: the `Language ID`, a text identifier (which is independent of the language) and the display text (which is different per language and text identifier). The user’s selection of the language will also be applied as a filter on this table. Instead of just typing the headline, I show the `DisplayText` for a specific text identifier.

Table 3-9. A table containing display texts for different parts of the report

Language ID	Textcomponent	DisplayText
EN	SalesOverview	Sales Overview
EN	SalesDetails	Sales Details
DE	SalesOverview	Verkaufsübersicht
DE	SalesDetails	Verkaufsdetails

Numerical content (e. g. values in different currencies)

Numbers are not strictly translated, but localizing in a broader sense also means, that numbers have to be converted between currencies. There are a

wide variety of solutions when it comes to finding the correct exchange rate. In a simple model you would have one exchange rate per currency (s. Table 3-10). In more complex scenarios you would have different exchange rates over time and an algorithm to select the correct exchange rate.

Table 3-10. A table containing exchange rates

Currency Code	Currency	Spot
EUR	Euro	1.0000
USD	US Dollar	1.1224
JPY	Japanese yen	120.6800

Data model’s metadata (e. g. the names of tables and columns)
Analytical database allow to translate the names of all artefacts of a data model (names of tables, columns, measures, hierarchies, etc.). When a user connects to the data model the preferred language can be specified in the connection string. Usually, only power users, who create queries and reports from the data model, care about this meta data. And usually they understand terms in English (or in the language the data model is created). Mere report consumers will not directly see any part of the data model, but only what the report exposes to them. And a report can expose text via

translated visual elements. Therefore, in my experience, the use case for meta data translation is only narrow.

User interface (e. g. Power BI Desktop)
You need to check the user documentation on how to change the user interface's language. In chapter 7 I will describe the settings for Power BI Desktop, Power BI Service and Power BI Report Server.

Some data sources expose their information in a way which looks like a table on first sight, but which – after a closer look – turn out to be not a classical table with information spread out over different columns. You will learn how to handle such tables in the next section.

Key-Value Pair Tables

You can see an example for a key-value pair table in Table 3-11. Such a table basically consists of only a key-column and a value-column (alas the name):

`Key`
This is the attribute. For example, "city".

`Value`
This is the attribute's value. For example, "Seattle".

Typically, you will find two extra columns:

`ID`

Common rows share the same ID. For example, for ID=1 there would be two rows, one for `key` =”name” and another one for `key` =”city”.

`Type`

This column contains a data type descriptor for the value column. As column `Value` must be of a string data type (as a string is the common denominator for all data types; a value of any data type can be converted into a string), `Type` tells us what kind of content to await in the `Value` column.

Table 3-11. A table containing key-value pairs of rows

ID	Key	Value	Type
1	name	Bill	text
1	city	Seattle	text
1	revenue	20000	integer
1	firstPurchase	1980-01-01	date
2	name	Jeff	text
2	city	Seattle	text

2	revenue	19000	integer
2	firstPurchase	2000-01-01	date
3	name	Markus	text
3	city	Alkoven	text
3	revenue	5	integer
3	firstPurchase	2021-01-01	date

Such a table is extremely flexible when it comes to adding new information. New information is simply added via an additional row (containing a new value for `Key` and its `Value`). No need to change the actual schema (column definition of such a table). This makes it very likable for application developers. Its flexibility is like storing information in flat files (JSON, XML, CSV, …).

On the other hand, it is very hard to build reports on top of such a table. Usually, you need to pivot the content of the `Key` column and explicitly specify the correct data type (e. g. to allow for calculation on numeric values).

There is though one single use case, where the table in its original state can make for very flexible reports. If the goal is to count the IDs on aggregations on different combination of keys to look for correlations, you can self-join the key-value pair table on the `ID` column. Then you filter the two `Key` columns individually (e. g. one on "name" and another on the "city"). Showing one `Value` on the rows and the other on the columns of a pivot table (or a *Matrix* visual in Power BI for that matter) and the count of the `ID` in the value's section. You get a quick insight into the existing combinations (e. g. that we have people of three different names living in two different cities and in which city how many people of each name live.). If you allow the report user to change the values for the two `Key` columns she can easily grasp the correlation of combinations of any attribute. You will see this in action in chapter 7.

Most reports you need to build are probably of a different nature: You need to group and filter some of the attributes and aggregate others. Therefore, you need to pivot all the keys and assign them to dedicated columns (with a proper data type), as shown in Table 3-12. Some reporting tools / visuals can do that for you. Most prominently Excel's pivot table or Power BI's matrix visual. They can pivot the key column for you, but they are not capable of changing the data type of the `Value` column. Aggregations will not be done at all or at least not in the proper way. Therefore, the best solution is one, where you prepare the pivoted table in the data model.

Table 3-12. The key-value pairs table pivoted on the key column

ID	name	city	revenue
1	Bill	Seattle	20000
2	Jeff	Seattle	19000
3	Markus	Alkoven	5

Who typically builds the data models in your organization: the domain experts or a dedicated (IT-)department? Both concepts have their advantages and disadvantages. The next section is dedicated to lay out their possibilities.

Combining Self-Service and Enterprise BI

We speak of *Self Service BI* when a domain expert (with no or little IT background) solves a data related problem on her own. This includes connecting to the data source(s), cleaning and transforming the data as necessary and building the data model, with no or just little code. The advantage is that involvement of IT is not necessary, which usually speeds up the whole process: All the requirements are clear to the person who implements the solution on her own. No infrastructure needs to be installed (everything runs on the client machine).

Everything available in an *Enterprise BI* solution, on the other hand, is built with heavy involvement of an IT department. Servers are set up. Code is developed. Automation is key. The advantage is that such a solution is ready to be scaled up and scaled out. All the requirements are implemented in one single place (on one of the servers). But this takes time to build. Sometimes collecting all the requirements and writing down the user stories for the engineers to implement can take longer than it would take for the domain expert to build a solution on her own.

No serious organization will trust business intelligence to be run on a client machine (*Self Service BI*), only. No serious domain expert is always patient enough to set up a project to implement a database and the reports (*Enterprise BI*). Therefore, the solution is to play both cards to the benefit of everybody.

Data needed for the daily tasks of information workers to be transformed into reports and ad hoc analysis should be available in a centralized data warehouse. Only here, *one version of the truth* can be made available. But there will always be extra data, which has not made it into the data warehouse (yet). That’s where *Self Service BI* has its place.

The question is, how to combine both worlds, so that the centralized data can be enriched with the extra data by the domain experts themselves. Chapter 7 will describe how this can be done in Power BI in a convenient way.

Key Takeaways

In this chapter I described real-world use cases. You learned about business problems and different concept of how to solve them. In later chapters you will learn how implement the solutions in DAX, Power Query and SQL:

- Binning of values can be done either with a simple lookup table (which contains all possible values and their bin) and a physical relationship between the values and the lookup table. Or you can describe the ranges per bin and apply a non-equi join between the values and the lookup table.
- New tables should only be added to a data model if the information cannot be added to an existing table. As a budget is usually on a different granularity than the actual data is, I took this as use case for a “multi-fact” data model.
- There are many aspects you must cover, when you want to implement localized reports: content of textual columns, text on the report, currency exchange rates, the names in the data model and the language of the user interface of the application you are working with.
- Both, *Self Service BI* and *Enterprise BI* will always exist side-by-side. In this chapter you learned about the challenges of how to combine both worlds. In chapter 7 you will see how both worlds can live together in Power BI.

Chapter 4. Performance Tuning

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sevans@oreilly.com*.

You are very blessed if performance tuning was never a topic in a report you built. Usually, it is not a question if, but only when the performance of a business solution becomes a priority. Generally, if you took everything we discussed in this book so far seriously, and transformed all data into a star schema data model, then you made an important step forward towards good performing reports. The shape of the data model plays an important role when it comes to performant reports. But, of course, there are many more pieces, which play a role in how fast reports will react to show the first data or react to filters. As this book is about data modelling, I will limit myself to

only discuss performance tuning topics around data modelling. My first computer had a Turbo button on the front part of its case, just next to the power button. I only used it rarely in the first weeks, but sooner or later I asked myself why I should run everything at a lower speed? The same applies to the data model you build. You should always build it with thoughts about good performance in the back of your mind, because, why should you build a model, where everything runs at a lower speed? Unfortunately, there is no Turbo button in Power BI you can just hit after you powered on the application. But there are some concepts you can apply. Read on to learn about them. If you are about my age, you maybe, in your youth, had a piece of paper with the telephone numbers of your close family, friends, and neighbors listed. Mine had the most important people first, and later I added more and more people to it. When the list got a decent length scanning through it every single time when I needed a number bored me. So, I started a new list and split the names and numbers on different papers: One paper per letter in the alphabet, with the numbers of all people with the same first letter of their first name together and the papers in alphabetical order as well.

This principle applies to databases as well: You can either create simple tables, where new data is (chronologically) added at the end of the table (or in between rows, after a row was deleted). It is very fast to add data to the table, because there is no need to find the right place, but just use the next empty space. But you pay a penalty when you read from the table, as the full content of the table has to be scanned every time you query it. Filters will only reduce the result set, but not the necessity of reading and evaluating every single row

(to find out if it satisfies the filter condition or not). The alternative is to store all the rows in a table in a certain order. As long as your filter is about the order key, finding the right rows can be faster: you ignore all the rows which do not fulfill the search condition, return all matching rows and stop as soon as non-matching rows appear in the table. In reality this can be even faster, as databases store extra information ("meta data") about the data just for the sake of making queries faster. But writing data into such a table will be a bit slower: The right position for the new rows has to be found. Maybe new space must made available at this position. And the meta data has to be maintained. The examples I've just mentioned should make clear the very important principle: You can exchange query speed with space on disk or in memory. Speeding up queries this way will likely slow down the write operations. In analytics it is most often the case that reading the data is done often and should go fast, while refreshes (= write operations) can be done on only scheduled points in time, and are fewer in numbers. Optimizing for read-operations is therefore a good idea, even when it slows down the write-operations. You can choose one of the following options as a way to store data in tables:

Only storing queries

You could opt for not physically storing (duplicating) data but keep the data in the original tables. Instead, you store only the query which will return the data in the desired shape. The advantage is, that no extra space is needed to store the (shaped) data and no steps to update the data has to be scheduled. The query result will always be fresh. Depending on the

type of transformations and the way the source tables are stored, the query will need some time to run.

Storing query results

Instead of running the queries against the data source every single time you need the data, you could store the result in a table and schedule a refresh. This will occupy space on disk or in memory, but speed up the queries, as the result of the transformations is already persisted. The challenge is to schedule the refresh often enough, so that the reports do not show stale data.

Adding meta data about the data

Here we can distinguish between meta data automatically added by the database system (like descriptive statistics about the data) and meta data explicitly added (like indexes). A database index is the same as the index at the end of this book. Instead of scanning (= reading) the whole book and looking for the term “foreign key”, you can jump to the index pages, where the important terms are ordered alphabetically. You will quickly find out, if the book talks about this term and find references to the book’s pages where you find more information about the term “foreign key”. While the book itself is “ordered” by its chapters, the index is ordered by an additional key. For tables it is not uncommon to have more than one index.

Adding data of different granularity to the data model

Querying a table by its sort order or over an additional index will be faster compared to not having such. But still, a query needs to collect the necessary information and calculate the aggregated values for the report (which typically does not show single transactions but data grouped by dimensions). Of course, it would be faster if those aggregated values are already stored (= persistsed) in a table. This is what aggregation tables are about: They store the identical information as the base table, but on different levels of granularity. For the report the table with the right level of aggregation will be used.

No matter which solution you want to implement, all of them are a strategy to exchange disk space (and therefore increasing the durtion of time to process the transformation and refresh the user-facing data model) with query runtime.

Key Takeaways

A good data model takes query performance into account. By following the principles of the earlier chapters, you already created a good performing data model. No matter which data model you design or which tools you are using, you have a wide variety of possibilities to control the performance, by applying a taste of two options:

- Directly querying the data source will always return the freshest information. But query time might not be acceptable (due to complex

transformation or a data source not designed for these ad-hoc queries).

- We can speed up queries by pre-computing all or parts of the data needed. Trans- formations can be persisted; statistics and indexes will help to find information faster and we can pre-aggregate data on different levels of granularity. This takes up extra space in the database and needs to be maintained regularly, so it does not contain stale data.
- By cleverly trading off query time and space used for the persisted data, you can achieve a balanced system, which satisfies the needs for fast reports and available storage resources.

With the next chapter we leave the world of concepts and dive into Power BI Desktop and its possibilities to create the data model, which will make the live of your report-creators easier.

Chapter 5. Understanding a Power BI Data Model

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sevans@oreilly.com*.

In this chapter you will learn how to create a useful data model in Power BI (and Analysis Services Tabular). This chapter will concentrate on the features of the *Model view*. The following parts of this book discuss options of bringing data of any shape into the desired shape in Power BI (which has been described in general in chapter 1). You will learn that Power BI needs a data model to work. I will go into details about the properties tables can have and how you can put them into relationships with each other. As you will find out, there is no need of explicitly marking primary and foreign keys, but you

still need to be able to identify them to make the relationships work. The cardinality of the relationships play an important role in Power BI. Lucky enough, you do not need to think about the joins and join path problems too much. You only need to create filter relationships for your data model. Power BI will automatically use these realtionships to join the tables appropriatly. Power BI will also make sure to execute the queries in a way, that the join path problems (described in Chapter 1) do not occure.

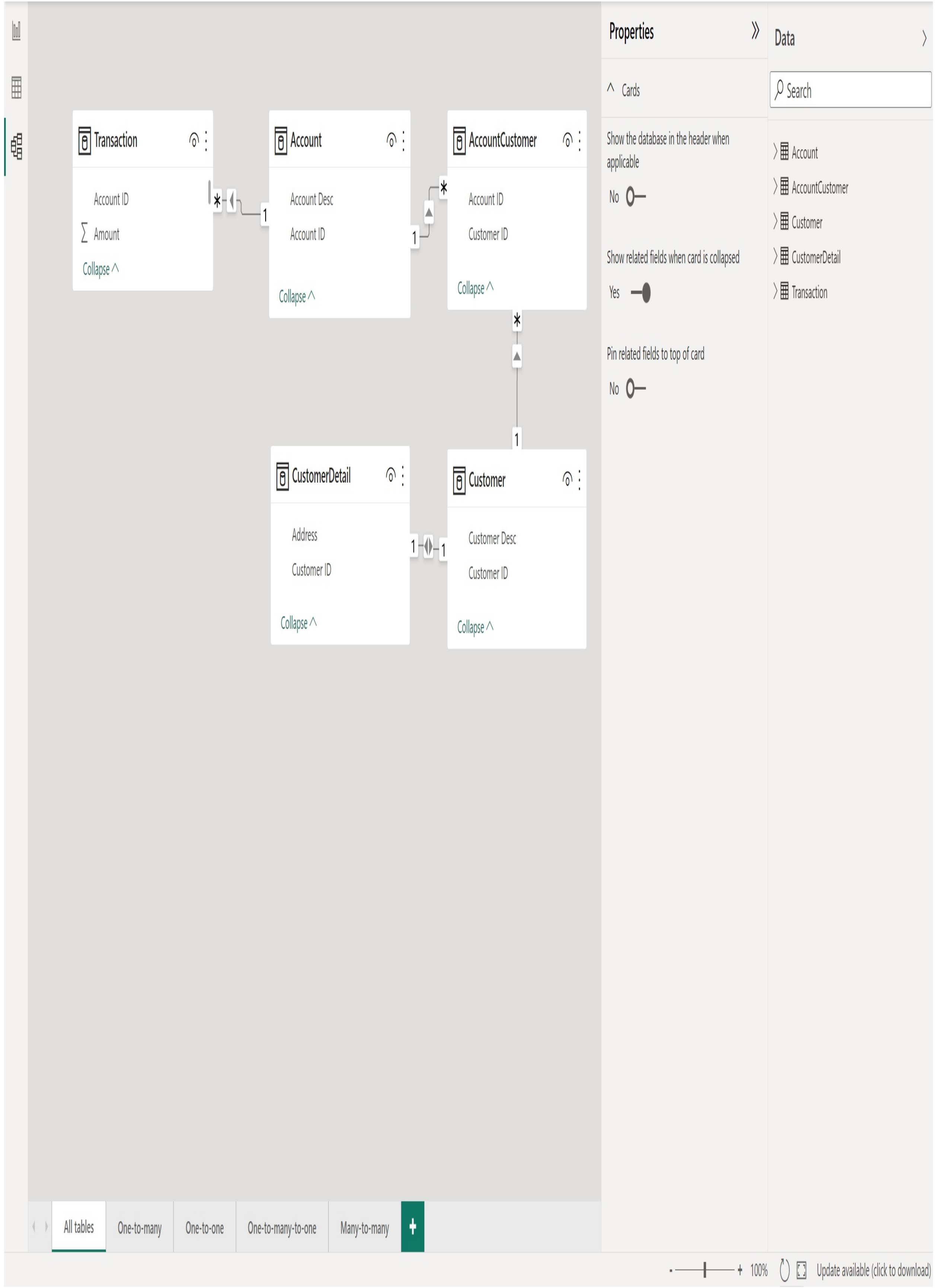
In this chapter I will, again, make clear why a single table is not a data model fit for Power BI and that a dimensional model is the go-to solution. Remember: The ultimate goal is to create a data model, which makes the report creator’s life easy.

Data Model

To get a visual overview and most of the options needed to create and modify the data model, you need to select the *Model view* in Power BI (or the *Diagram View* in Visual Studio in case of Analysis Services Tabular). This view looks much like an Entity-Relationship-Diagram (ERD), but has subtle differences, we will discuss during this chapter. The tab named *All tables* shows each and every table in the data model, as shown in Figure 5-1. For bigger data models (read: data models with a lot of tables) it makes sense to create separate layouts for only selected tables of your data model. This will give you a more manageable view for different parts of your data model. For

example, if your data model contains several fact tables, it might be a good idea to create a separated view per fact table (and all the connected dimension tables), as an alternative to the *All tables* view. While in the *All tables* view not all tables might fit on your screen (or only, if you zoom-out so much, that you can’t read the names of the tables and columns anymore), a separated layout with less content can be helpful.

You can create such a separate layout by clicking on the +-symbol, just right of *All tables*. Then, you can add individual tables from the fields pane (on the very right of the screen) via drag-and-drop. Alternatively, you can right click a table’s name (either in the model’s canvas or in the fields pane) to add not only the table itself, but all the tables with a relationship to it, as well (*Add related tables*).



Transaction
Account ID
Amount
Collapse
Account
Account Desc
Account ID
Collapse
AccountCustomer
Account ID
Customer ID
Collapse
CustomerDetail
Address
Customer ID
Collapse
Customer
Customer Desc
Customer ID
Collapse
Properties
Cards
Show the database in the header when applicable
No
Show related fields when card is collapsed
Yes
Pin related fields to top of card
No
Data
Search
Account
AccountCustomer
Customer
CustomerDetail
Transaction
All tables
One-to-many
One-to-one
One-to-many-to-one
Many-to-many
100%
Update available (click to download)

Figure 5-1. Model View

The *Model view* has three properties:

- You can decide to *Show the database in the header when applicable*. This is applicable in data models in *DirectQuery* mode and *Dual*. You will learn about the different storage modes in chapter 8. This setting is turned off by default.
- Each table can be either *expanded* to show every column of the table or be *collapsed*. When a table is collapsed, no columns are shown, unless you ask to *Show related fields when card is collapsed*. Related fields are columns, which are used in relationships. This property is enabled by default.
- As you might guess, if you *Pin related fields to top of card*, columns being part of a relationship are shown on top of the field list. By default, this setting is disabled, and all fields are shown in alphabetical order (measures are listed after the columns, again in alphabetical order).

Tables play an important part of the data model. Let’s look on their properties in the next section.

Basic Concepts

Tables

Every rectangle in the *Model view* represents one table in the data model. In the header you can read the table’s name. Below, the columns (= fields) are listed, in alphabetical order.

A table offers a list of functionalities, which you can access by either right-clicking the table name or by left-clicking the ellipses (…), as I did in Figure 5-2.

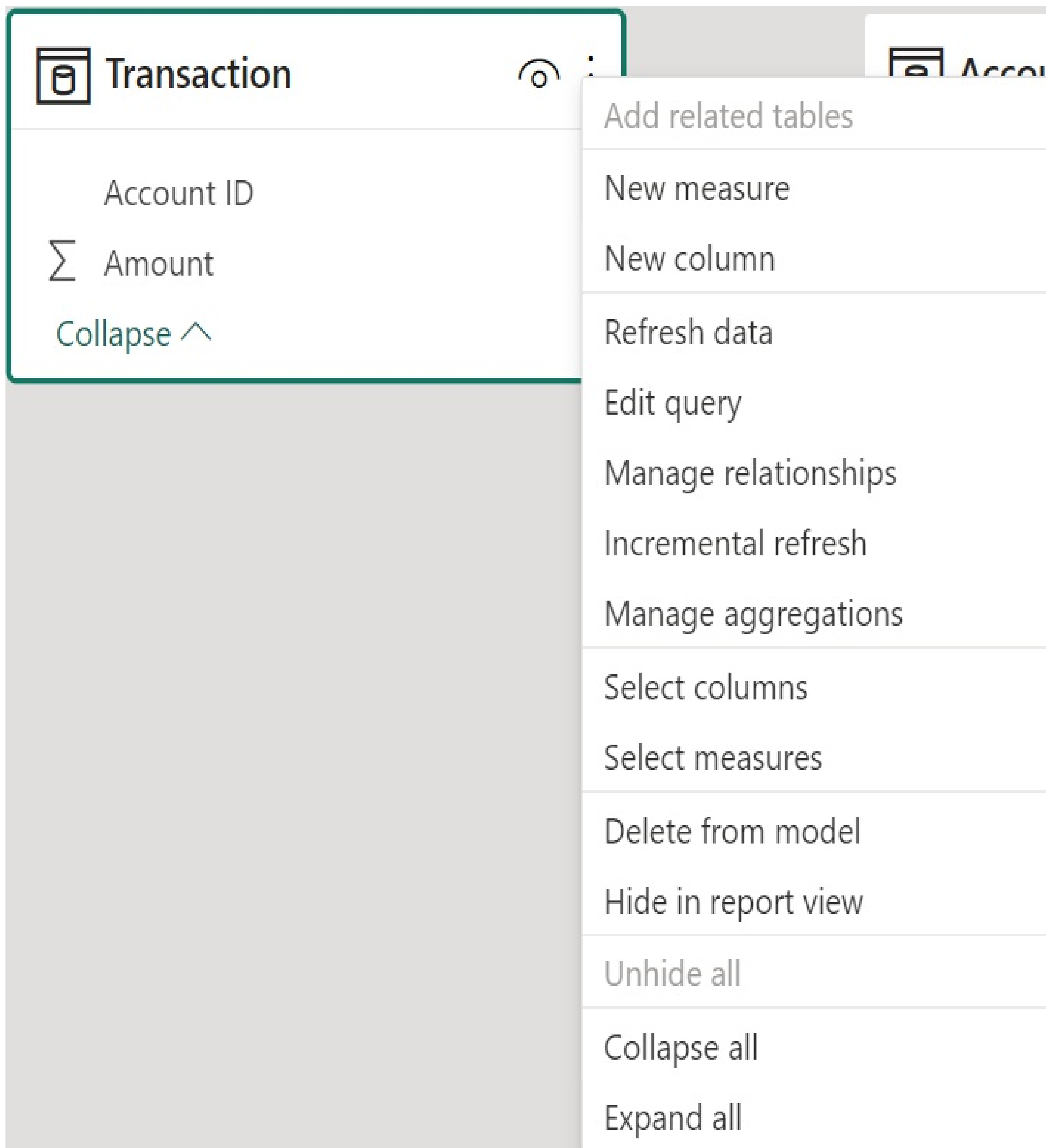


Figure 5-2. Model View Context Menu

You have many options, when it comes to :

- *Add related tables* will add all tables which have a filter relatinship with

the chosen table to the model view. This option is only available, when the table has relatinships to other columns and only in a layout view (not in *All tables*).

- You can create a *New measure* or a *New column* within the selected table. In Chapter 6 you will learn more about these two options.
- You can choose *Refresh data* to refresh the content of the whole table. The data source has to be available at this point in time.
- *Edit query* will open the *Power Query* window. In [Link to Come] I will introduce to the capabilities of *Power Query*.
- In "Relationships" you will learn everything about the options to *Manage relatinships*.
- [Link to Come] covers all the details of *Incremental refresh* and *Manage aggregations*.
- With *Select columns* you can select all columns in this table. You can then change the properties for all of them in one go.
- On the other hand, *Select measures* marks all measures in this table. You can then change the properties for all of them in one go.
- If you choose *Delete from model*, the whole table will be removed not only from the layout view, but from the whole file (incl. Power Query).

WARNING

Be careful, this step cannot be undone. Make sure to save intermediate versions of your changes to have a backup.

- With *Hide in report view* the table (and all its columns) are hidden. The goal is to not overwhelm the report creators with tables (or columns) holding intermediate results, not intended to be shown in a report.

WARNING

A hidden table (or column) can still be seen by the user if the user enables *View hidden*. Therefore, this is not a security feature: Don't use it to hide tables with sensitive content. Use it to hide tables, which are only helping to build the data model, but which do not contain any content, which should be used in a report (e. g. to show the value, to filter on it, etc.).

- The option *Remove from diagram* is only available in layouts, but not in tab *All tables*. This does not remove the table from the whole file, but just from the current layout.
- With *Unhide all* you can disable the property *Hidden* for all elements within this table. Again, this step cannot be undone. In case of a mistake, you need to hide the individual columns again.
- *Collapse all* collapses the height of the table to only show column which are used in relationships (or no column at all, depending on the overall settings).
- *Expand all* expands the height of the table to its original size.

In Power BI and Analysis Services Tabular a table can contain not only columns, but measure and hierarchies as well. Measures are written in the DAX language. Chapters 9 to 12 will show you many capabilities of the DAX language and in chapter 10 you will find more information about

measures in the section about calculations. Hierarchies group several columns into one entity. There is a whole section dedicated to hierarchies in chapter 6.

Tables have properties in Power BI, as you can see in Figure 5-3:

Properties

General

Name

Transaction

Description

Enter a description

Synonyms

transaction

Row label

Select a row label

Key column

Select a column with unique values

Is hidden

No

Is featured table

No

Edit

Advanced

Storage mode

Import

Figure 5-3. Model View Table Properties

Name

This is the name of the table. You can change the name either here or by renaming the Power Query associated with this table in the Power Query windows (which I discuss in chapter 13).

Description

The description is shown in a tooltip when you (or the report creatore) move the mouse cursor over the table’s name in the *Data* list. Typically I use this field to include a few sentences to describe the content of the table.

Synonyms

This field is automatically propagated by Power BI as a comma separated list. You can add your own synonyms as well. The list of synonyms helps the Q&A (question and answer) visual to answer your questions. You should type in alternate terms used in your organization (e. g. some people may use revenue as an alternate name for table sales; you would then enter *revenue* as a synonym for sales). Visuals are out of scope for this book. You will find an in-depth description of the Q&A visual in my book “Self-Service AI with Power BI Desktop”, though.

Row label

You should select the column containing the name of the entity as the *Row*

label of the table. This feature helps the Q&A visual when you reference a table to select the column to show. It helps in Excel in a similar way, too.

Key column

You should select the column containing the primary key as the *Key column* of a table. Again, this feature helps the Q&A visual.

Is hidden

Enabled means that the table is hidden by default, *disabled* means that the table is shown by default. You should hide all tables which do not contain content relevant for the reports, but which are needed to create the data model in the right shape, e. g. *bridge* tables.

WARNING

The user can still *View hidden* elements. Therefore, keep in mind that this is *not a security feature*.

NOTE

Technically, tables cannot be hidden in Power BI or Analysis Services Tabular, but columns only. If a table contains hidden columns only, then the table is hidden as well. So, hiding a table, changes the *Is hidden* property of all columns within the table. If you choose to unhide a table, all columns will be visible (no matter if they have been hidden or not before you changed this setting on the table level).

Is featured table

Makes the table easier to find in Excel, when using the Organization Data Types Gallery (*https://learn.microsoft.com/en-us/power-bi/collaborate-share/service-create-excel-featured-tables*).

Storage Mode (Advanced)

The storage mode of a table can be either *Import*, *DirectQuery* or *Dual*. You will learn more about using the storage mode to your advantage in chapter 8.

Columns reside in tables and have properties as well, as you can see in Figure 5-4. In the *Model view* you set the following properties:

Properties

General

Name

Amount

Description

Enter a description

Synonyms

amount

Display folder

Enter the display folder

Is hidden

No

Formatting

Data type

Whole number

Format

Whole number

Percentage format

No

Thousands separator

No

Decimal places

0

Advanced

Sort by column

Amount (Default)

Data category

Uncategorized

Summarize by

Sum

Is nullable

Yes

Figure 5-4. Model View Column Properties

Name

This is the name of the column. You can change the name either here or in the Power Query window. You will learn about Power Query in [Link to Come]>>.

Description

The description is shown as a tooltip, when you hover over the column in the fields list in the *Data* blade. Typically I add a few sentences to describe the content of the column or the formula of a DAX measure.

Synonyms

The provided synonyms are used by the Q&A visual to find the columns not directly referenced by their (technical) name, but by alternative names and versions as well (e. g. a user might ask Q&A about *revenue*, but the column is called `Sales Amount`; you would then add `revenue` in the list of synonyms for the column `Sales Amount`).

Display folder

In tables with a lot of elements, browsing through the list can be tedious. Putting a column (or a measure) into a folder provides some structure. To put the column `Sales Amount` into a folder called `Sales` you would just enter “Sales” into the *Display folder*. You can even create subfolder by using the backslash (“\”) in this field. Etc. “KPI\Sales” puts the column

into a folder `KPI` and there into a subfolder `Sales`.

Is hidden

Hides the column. You should hide all columns which we need for the data model only (e. g. all keys), but should/will never be shown in a report.

WARNING

The user can still *View hidden* elements. Therefore, keep in mind that this is *not a security feature*.

Data type (Formatting)

Every column has a dedicated data type. I discuss the available data types later in this section.

Format (Formatting)

You can choose a format, in which the value of the column is shown. The available formatting options are dependent on the data type, of course: Percentage format, Thousands separator, Decimal places, Currency format, etc.

NOTE

Changes to the format do not change the data type (= internal storage) of the column. For example, if you want to get rid of the time portion of a timestamp, you could change the format to only showing the date portion. But then the time portion is still stored in the data model (which is expensive in terms of storage space, as you learned in chapter 2 and might lead to non-working

filters, if the dimension table does not contain the time portion, but the fact table does, for example). If nobody will ever report on the time portion, then it is a good idea to change the data type to *Date* instead.

Sort by column (Advanced)

By default every column is sorted by the value it contains. In some situations, this can be unpractical. Typically, you do not want a list of month names sorted alphabetically, but by their month number. This is what this property is for. You would select column `Month Number` as the *Sort by column* of `Month Name`. You can also use this option to show names of categories or countries in a specific order. For every value of the column, only a single value of the *Sort by column* must be available; you cannot sort the `Month Name` by the `Date` or by a month key, which contains both, the year and the month number. In other terms, the relationship between the column and the *Sort by column* must be of a one-to-one or a many-to-one cardinality but can't be a one-to-many or many-to-many.

Data category (Advanced)

Assigning a *Data category* to a column allows Power BI to default to a certain visual, when this column is used. E. g. if you mark a column as a *Place*, Power BI will suggest showing the content on a *Map visual*.

Summarize by (Advanced)

This setting is intended to help newcomers to Power BI's data modelling

capabilities, to allow to add a numerical column into a visual, where the content should be aggregated. For example, if you add the `SalesAmount` column to a visual, you usually do not want to get a (long) list of all the rows of the `Sales` table showing the individual `SalesAmount` value, but you want to sum up the numbers to a total. *Summarize by* allows you to specify, how the values are aggregated: *Sum*, *Average*, *Min*, *Max*, *Count*, and *Distinct Count*. For numeric values, which you do not want to aggregate (e. g. the year or the month number), you must specify *None*.

Any setting different from *None* will create a so-called implicit measure in the data model, containing the chosen aggregate function. Unfortunately, implicit measures do not work when you use Excel to connect to this data model. Implicit measures will also stop working as soon as you create *Calculation groups* (you will learn about them in chapter 10 in the section about calculations in DAX).

The solution is to explicitly create measures for columns, where you need aggregations to be applied, and set *Is hidden* to *Yes* for those columns. (You will learn more about explicit measures in chapter 10 as well).

Is nullable (Advanced)
Specifies, if the column may contain blank (or null, as it is called in relational databases). If you consider blank in this column as a data quality issue, then you should turn this setting to *No*. This will lead to an error

during the refresh in case a row actually contains blank for this column. Every row of a column must conform to the column's datatype in Power BI. Let's take a closer look onto the different data types, Power BI allows to choose from:

Binary

This data type is not supported and exists for legacy reasons only. You should remove columns of this datatype before loading into Power BI and Analysis Services or just delete them from the model in the *Model view*.

True/false

A column of this type can contain Boolean values: True, or false. But this data type is no exception in the sense that it additionally can also contain *blank*. *Blank* represents an unknown value. In a note below I will provide you more information about *blank*.

Fixed decimal number

This datatype can contain numbers with up to four decimals and up to 19 digits of significance. You can store values between -922,337,203,685,477.5807 and +922,337,203,685,477.5807. These 19 digits are identical to the *Whole number*, as a *Fixed decimal number* is stored in the same internal format, but with the extra information, that the last four digits are decimals. For example, the *Whole number* 12345 and the *Fixed decimal number* 1.2345 are stored in the exact same internal representation, with the only difference, that the *Fixed decimal number* is

automatically divided by 1000 before it is shown. Due to the limit to four decimal places, you can face rounding error, when values are aggregated.

Decimal number

Is 64bit floating point number which can handle very small and very big numbers, both in the positive and negative spectrum. As it is only precise to up to 15 digits, you may face rounding error, when values are aggregated.

Date/time

Represents a point in time, precise to 3.33 ms. Internally in a database all date and time related data is stored as a decimal number counting the days since an arbitrary point in time. The decimal portion represents the parts of the day (e. g. 0,5 represents 12 PM). I am pointing this out to make sure that you do not make the mistake of thinking that a date/time is stored in a certain format (e. g. “August 01 2023 01:15:00 PM” or “2023-08-01 13:15:00”). As already pointed out, the *Format* properties task is to put a value into a user-friendly format we, as humans, can read, but does not change the internal representation (which is 45,139.55 in Power BI for the given example – and would obviously be not very user-friendly to show in a report).

Date

Represents a point in time, without the time portion. Everything mentioned for data type *Date/time* also applies here. Internally this data

type is represented as a whole number (e. g. 45,139 to represent August 1, 2023).

Time

Represents a point in time, without the date portion. Everything mentioned for data type *Date/time* also applies here. Internally this data type is represented as a decimal number with only the decimal portion (e. g. 0.55 to represent 1:15 PM).

Text

Holds unicode character strings, which can hold up to 268,435,456 characters.

Whole number

Values of this data type are stored as an 64bit integer value. This data type does not have decimal places and allows for 19 digits. It covers the spectrum between 9,223,372,036,854,775,807 and + 9,223,372,036,854,775,806.

NOTE

For databases it is typical that every data type supports also an “unknown” value. In relational databases and in Power Query this is represented by *null*, in DAX by *blank*. It is important to understand that this unknown value is different from an empty string, the numeric value zero (0) or a date value of January 1 1900. The information that something is unknown might be an important fact (and should not be set equal to some default value). In a user-friendly data model, an unknown value should be replaced by something which explicitly tells that the value is unknown (e. g. string “N/A” or “Not available”), as you already learned in chapter 2.

Tables don’t live just by themselves, but usually contain information which is somehow in relation to information in other tables. Read on to learn more about this kind of relationships.

Relationships

Relationships in Power BI connect tables with each other and *look* like foreign key constraints in a relational database, but *work differently*. While foreign key constraints limit possibilities (they prohibit having a value in a foreign key column, which cannot be found in the related primary key column), relationships in Power BI exists solely to propagate a filter from one table to another. If you filter the *Date* table on a certain year, the filter relationship will propagate this filter to the *Sales* table, so queries only show sales for the specified year. Their effect enables what we usually perceive as something very natural. But for this natural thing to happen, we must help Power BI by setting the relationships right.

Creating filter relationships is rather easy. Power BI offers three methods, which all lead to the same result:

Automatic creation
Power BI can automatically create and maintain filter relationships for you, when loading new tables. Under *File – Options and Settings –*

Options – Current File – Data Load you can find three options related to *Relationships* (s. Figure 5-5). You can Power BI let import the relationships from a data source on first load (when the data source is a relational database and the foreign key constraints are available). You can let Power BI maintain those relationships when refreshing data. And Power BI can also autodetect new relationships after data is loaded (by applying a set of rules: the column names must be the same, the data types of these columns must be the same, and in at least one of the two tables the column’s value must be unique).

WARNING

If your source system follows the rule to name all primary keys e. g. “ID”, then the automatic detection of relationship will end in a relationship chaos, as Power BI will most probably start creating relationships between all those columns. Either turn this feature off or change the naming convention to add the table’s name to the key fields (“ProductID” instead of just “ID”, or similar).

Options

GLOBAL

- Data Load
- Power Query Editor
- DirectQuery
- R scripting
- Python scripting
- Security
- Privacy
- Regional Settings
- Updates
- Usage Data
- Diagnostics
- Preview features
- Auto recovery
- Report settings

CURRENT FILE

- Data Load
- Regional Settings
- Privacy
- Auto recovery

Type Detection

- [x] Detect column types and headers for unstructured sources

Relationships

- [x] Import relationships from data sources on first load ⓘ
- [ ] Update or delete relationships when refreshing data ⓘ
- [x] Autodetect new relationships after data is loaded ⓘ

Learn more

Time intelligence

- [ ] Auto date/time ⓘ Learn more

Background Data

- [ ] Allow data previews to download in the background

Parallel loading of tables ⓘ

Maximum number of concurrent jobs Learn more

- (•) Default
- ( ) One (disable parallel loading)
- ( ) Custom

Q&A

- [x] Turn on Q&A to ask natural language questions about Learn

OK Cancel

Figure 5-5. Options – Curent File – Data Load

Drag and drop

A very simple way of creating a relationship is to drag one column over another (from a different table) in the *Model view*. In theory, it does not matter which of the two columns you drag over the other one. The cardinality and filter direction (s. below) are automatically set for you. It is though always a good idea to double-check if all properties of the created relationship are as they should be. I’ve seen it more than once, that Power BI created a many-to-many (due to unintended duplicates in a table) or a one-to-one relationship (due to uncomplete test data with only e. g. one order per customer), where it should have been a one-to-many cardinality instead.

Dialog box

Via the ribbon you can choose *Home – Manage relationship* to open a dialog box from which you can create a *New relationship*, start *Autodetect* (as described in the paragraph before), *Edit* or *Delete* an existing relationship (s. Figure 5-6). By clicking on the checkbox *Active* you can (de-)activate a relationship. I explain this feature below ([Link to Come]).

TIP

The order the relationships are shown looks unpredictable to me: A relationship between the *Date* and the *Sales* tables might show up with the *Date* table first (and ordered by it) or with the *Sales* tables first (and ordered by that). If you cannot find the relationship you are looking for, double-

check if can find it listed with the other table first in this dialog box.

Manage relationships

Active	From: Table (Column)	To: Table (Column)
✓	AccountCustomer (Account ID)	Account (Account ID)
✓	AccountCustomer (Customer ID)	Customer (Customer ID)
✓	CustomerDetail (Customer ID)	Customer (Customer ID)
✓	Transaction (Account ID)	Account (Account ID)

New... Autodetect... Edit... Delete

Close

Figure 5-6. Modeling – Manage Relationships

TIP

As filter relationships are so important, I would not rely (only) on automatic creation. Even if you let Power BI create the relationships for you in first place, I would make sure to review every single one, to assure that no relationship is defined in a wrong way (read: check the cardinality), that no relationship is missing and that no unnecessary relationship was created.

Figure 5-7 shows you the property pane in the model view and the *Edit relationship* dialog for the same relationship.

Edit relationship

Select tables and columns that are related.

Transaction

Account ID	Amount
11	1000000
11	2000000
12	5000000

Account

Account ID	Account Desc
11	Bill
12	Jeff
13	Bill - Jeff

Cardinality: Many to one (*:1)

Cross filter direction: Single

☑ Make this relationship active

☐ Apply security filter in both directions

☐ Assume referential integrity

OK Cancel

Properties

^ Relationship

Table	Column
Transaction	Account ID

Cardinality: Many to one (*:1)

Table	Column
Account	Account ID

Make this relationship active

Yes

Cross filter direction: Single

Apply changes

Open relationship editor

Figure 5-7. Model View Relationship Properties

A filter relationship in Power BI consists of the following properties:

First table

Sometimes also described as the left table. It is one of the two tables, for which you create a relationship. Which table the first/left table is, is not important, as any one-to-many relationship can also be seen as a many-to-one relationship – they are identical.

Column in the first table

You can click on the column name shown for the first table to select one or choose from the listbox. The *Edit relationship* dialog windows shows a preview of the values of the first three rows. Selecting more than one column is not possible, as Power BI does not allow to use composite keys. In cases where you must work with composite keys, you need to simply concatenate the columns with a separator character in-between (as a DAX calculated column, in Power Query, SQL or the data source) before you can create the relationship.

Second table

Sometimes also described as the right table. It is one of the two tables, for which you create a relationship.

Column in the second table

You can click on the column name shown for the second table to select

one. The *Edit relationship* dialog window shows a preview of the values of the first three rows. Selecting more than one column is not possible.

Cardinality

Cardinality describes, how many rows in the other table can maximally be found for a single row of one table. Chapter 1 keeps you fully covered on this topic.

Cross filter direction

As explained, the sole purpose of a relationship in Power BI is to propagate a filter from one table to another. A filter can go in either direction, or even in both directions. I strongly advise sticking to the best practice of only using single-directed filters. These filters are propagated from the one-side of relationship to the many-side of a relationship. Other filter directions (especially the bi-directional filter) might lead to ambiguous data models, which Power BI will prohibit you to create, and/or poor report performance.

WARNING

Bi-directional filters are sometimes used to create cascading filters (where a selection of a year limits the list of products in another filter to only those where there have been sales in the selected year). I strongly advise you to solve this problem through a filter in the slicer visual instead: just add e. g. the *Sales Amount* measure as a filter to the slicer visual and set the filter to *Is not blank*. Now any filter's result will cascade into this slicer. Repeat this for every slicer visual.

Make this relationship active

Only one single active relationship can exist between the same set of two tables in Power BI. The first created relationship between two tables is active by default. If you create additional relationships between these two tables, they can only be marked as inactive. In the model view you can distinguish active and inactive relationships by how the line is drawn: Active relationships are represented by a continuous line, while inactive relationships are drawn as a dashed line. In chapter 9 you will learn how you can make use of inactive relationships with the help of DAX. In chapter 6 in the section about *Role-playing dimensions* I will show you alternatives to having more than one relationship between two tables.

Apply security filter in both directions

This setting is only relevant, if you have implemented *Row Level Security* (RLS) in the data model and use bidirectional filters (which is not recommended; s. above "Cross filter direction") . Propagation of *Row Level Security* is always single-directed (from the table on the one-side to the table on the many-side), unless you activate this setting. Learn more about RLS in Microsoft's online documentation (*https://learn.microsoft.com/en-us/power-bi/enterprise/service-admin-rls*).

Assume referential integrity

This property is only available when using *DirectQuery* (you learn about *DirectQuery* in chapter 8). Activating this checkbox will let Power BI assume that the columns used for this relationship have a foreign key

constraint in the relational database. Therefore, Power BI can use inner joins (instead of outer joins) when querying the two tables in one single query. Inner joins have a performance benefit over outer joins. But with inner joins, rows could be unintentionally filtered out, when referential integrity is violated by some rows, as you learned in chapter 1.

Independent of these relationship settings, joins in Power BI are always outer joins (except for *DirectQuery*, when *Assume referential integrity is enabled*). This guarantees under all circumstances, that no rows are unintentionally lost (even when referential integrity in the data is not guaranteed). Missing values are represented as *Blank*.

The Power BI data model does not allow for non-equi joins. In chapter 11 the section on *Binning* will teach you ways of implementing non-equi joins with the help of DAX.

Relationship consists of both a primary key and a foreign key. Let's start how Power BI handles primary key in the next section.

Primary Keys

In Power BI you do not explicitly mark primary keys (except when using DirectQuery to benefit from a better report performance). Implicitly, any column used in relationships on the one-side is a primary key. If the column on the one-side contains duplicated values, then the refresh will fail. Empty

or blank values for a column on the one-side are allowed. I strongly encourage you to make sure in the ETL to have no blank values anywhere, neither in primary keys nor other columns, but to replace them with a meaningful value or description (like “not available”). In [Link to Come] you learn different ideas of how to achieve this.

Power BI’s data model does not allow composite keys. In case you decided for a composite key you need to concatenate all the columns participating in the key into one single column (usually of type *Text*). Make sure that you add a separator character between the values. Look for a (special) character which will never be part of any of the column’s values (e. g. a pipe symbol | when concatenating names). This makes sure that the result of concatenating “ABC” and “XYZ” is different from concatenating “ABCX” and “YZ”. With the separator you get “ABC|XYZ” in one case and “ABCX|YZ” in the other. Without the separator you would end up with the identical primary key “ABCXYZ” for both rows, which is problematic as Power BI can then not distinguish those different two rows from each other.

Surrogate Keys

As a relationship can only be created on a single column, Power BI does not allow composite keys. I strongly advise to use columns of type *Whole number* for the relationships, as they can be stored more efficiently (compared to the other data types) and will therefore make filter propagation happen faster (which leads to faster response time in the reports). While the

key of the source system could be of any type, a surrogate key is usually a *Whole number*. This makes them perfect keys for Power BI. Learn more about creating surrogate keys in [Link to Come].

An important reason to have primary keys is, to reference a single row in this table. The column in the referencing table is called a foreign key. Learn more about it in the next section.

Foreign Keys

You do not explicitly define foreign keys in Power BI. They are implicitly defined when you create relationships. A column on the many-side of a relationship is the foreign key.

In case you decided for a composite key as a primary key you need to concatenate all the columns participating in the foreign key as well. Make sure that you concatenate the columns in the very same order as you did for the primary key and to use the same separator character.

When it comes to primary and foreign keys, you should be prepared to know how many rows in the table containing the primary key are a available for a single foreign key, and the other way around. This is called *Cardinality* and covered in the next section.

Cardinality

For every relationship you also need to specify its cardinality. Power BI offers three types of cardinalities:

- One-to-many (1:m, 1-*)
- One-to-one (1:1, 1-1)
- Many-to-many (m:m, -)

All relationships in Power BI are automatically conditional on both sides. That means that it is allowed, that a corresponding row in the other table is not available. For Power BI it is OK, when there is no row in the `Sales` table for a specific `Customer`. This is also OK in the real-world, as a brand-new customer might not have ordered yet. But it is also OK for Power BI if no customer can be found for a `CustomerID` in the `Sales` table. In the real-world this would be an issue in most cases: Either no `CustomerID` was stored for a sale. Then you need to clarify with the business if this is indeed possible (for edge cases). Or the `CustomerID` provided in the `Sales` table is invalid. That would be a data quality issue you would need to dig deeper into. Because if the `CustomerID` is invalid for a row, who knows if the `CustomerID`'s for the other rows are just valid by random, but contain the wrong information?

WARNING

Keep in mind, that Power BI will create a many-to-many relationship not only for the classical many-to-many relationships (e. g. one employee works for many different projects, one project has many employees), but in all cases, where none of the two columns used for creating the relationship only contains unique values. In case of data quality issues (like if there is duplicated customer row or you

have more than one row with a blank `CustomerID`), Power BI will not let you to change the relationship to a one-to-many.

Relationships of many-to-many cardinality are called “weak” relationships, as they come with two special effects, which are reasons why you should avoid this type realtionships (and using a bridge table instead, as discussed in “Types of Tables”):

- In case of missing values in the dimension table or wrong foreign keys in the fact table, no blank rows are shown in the reports to represent the values for the missing/wrong keys. Instead, these values are not shown at all. Reports and the totals might show incomplete numbers. This effect only hits you in case of data quality issues. Making sure that there are no missing rows in the dimension table and that there are no invalid foreign keys in the fact tables is a good idea anyways.
- Calculations in DAX which use function `ALL` (or `REMOVEFILTERS`) to remove filters will not remove filters on tables connected over a weak relationship. This can be a trap when you ask to *Show value as - Percent of grand total* or when creating more complex measures in DAX. As report creators can create measures (containing function `ALL` in their expression), this problem can appear anytime and can only be avoided by avoiding many-to-many relationships. That’s why I try to avoid many-to-many relationships. Find explanations about the specialties of many-to-many relations ships at *https://learn.microsoft.com/en-us/power-*

bi/transform-model/desktop-many-to-many-relationships#use-a-relationship-with-a-many-to-many-cardinality-instead-of-the-workaround. In case of very large tables, many-to-many relations might have a better performance compared to the solution with a bridge table, though.

In the *Model view* you can easily spot such problematic relationships: The line drawn for a many-to-many filter relationships shows gaps on both sides (as you can see in Figure 5-8). The solution to avoid all these effects is to model a many-to-many relationship via a "bridge" table. You will learn this technique later in "Types of Tables".

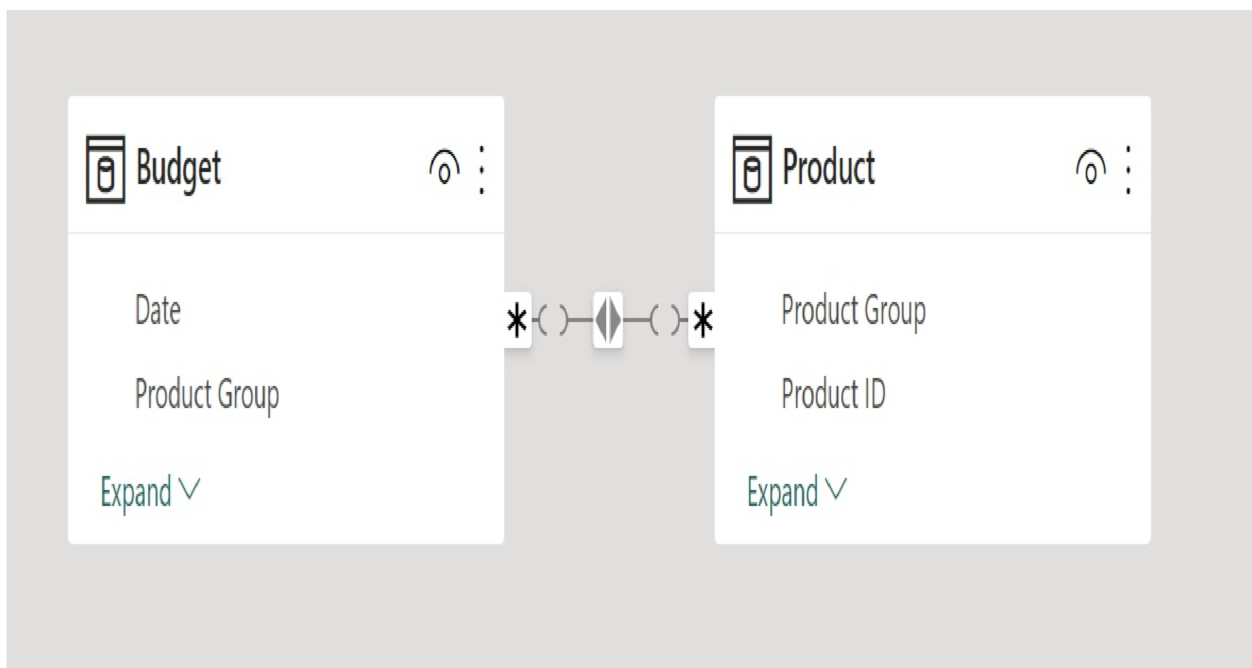


Figure 5-8. Many-to-many relationships are "weak" relationshps, drawn with gaps at both ends.

TIP

Always make sure that you understand how the relationship between two entities are in the real world.

In cases you can’t set the configuration for a relationship in Power BI accordingly, double-check the reason for that. Don’t just carry on but clarify the business’ requirement and the data quality.

Combining Tables

Set Operators

Set operators are not available directly in the data model of Power BI, but when querying data only. Jump to the other parts of this book to learn about if and how to implement set operators in DAX (chapter 9), Power Query / M (chapter 13), and SQL (chapter 17). On the other hand, joins are a regular thing when working with data models in Power BI. Read on, to learn more about that.

Joins

Joins are implemented implicitly over the data model’s relationships. The filter relationships stored in the data model, are implemented automatically by the Power BI’s storage engine to perform necessary joins. That means, if you create reports or write queries in DAX, there is no need to (explicitly) specify a join operator, as this is implicitly done by the storage engine for you (and the report user). The filter relationship defines the join predicate (= which two columns are involved). The predicate is always done as an equi-join (= the values of the two columns must match). You cannot define non-

equi joins in Power BI’s data model. In chapter 9 you will learn how to implement queries for non-equi joins with DAX. You can also perform non-equi joins in Power Query / M and SQL, to join queries and load the result as one single table into Power BI.

The natural join is simulated by Power BI’s ability to create the relationships in the data model automatically for you. If the filter relationship is not explicitly defined in the data model, then no natural join will happen, when creating reports.

Unfortunately, you cannot create self-joins at all. In the section about hierarchies (in chapters 10, 14, and 18), you will learn how to flatten parent-child hierarchies, so you are able to report on such hierarchies.

By default, all joins in Power BI are implemented as outer joins. This guarantees that no rows are lost, even when referential integrity of the model is not guaranteed. In relational databases outer joins come with a performance penalty (compared to inner joins). The storage engine behind Power BI was always built with outer joins in mind – so there is no performance penalty to be expected. And there is also no way of comparing, as you can’t execute inner joins on data imported into Power BI. When you don’t import data, but use DirectQuery (on relational data sources), then it is recommended that you first, guarantee that referential integrity is in place in the data source and second, you can tell Power BI so (with the table’s property in the model view). Then the storage engine will use inner joins instead of outer joins

when querying the data source (and making use of the performance advantage).

Joins are necessary to bring information, spread over several tables, back into the result of a single query. Combining the tables in a query can be tricky – but Power BI covers the usual problems for you, as you will see in the next section.

Join Path Problems

No worries, Power BI got you covered on all join path problems: None of the three problems discussed in Chapter 1 (loop, chasm trap, and fan trap) are an issue in Power BI. But you can see this for yourself in the following list:

- You cannot create a *loop*, neither directly nor indirectly (via intermediate tables), as Power BI will not allow you to create multiple active paths. Power BI will force you to declare such a relationship as inactive.
- Power BI has implemented a logic to avoid the negative effects of a *Chasm trap*. The example in Figure 5-9 shows a report with three table visuals. The table on top left shows the reseller sales per day. Just below you see the internet sales per day. On the right, the results of both tables are combined, per day. The two tables have each a one-to-many relationship to the `Date` table (and therefore a many-to-many relationship between themselves). As you can see, none of the sales amount for a day (or for the total) is wrongly duplicated, but matches the

numbers shown for the individual results.

FactResellerSales

DateKey	SalesAmount
20101201	$489,329
20110101	$1,538,408
20110301	$2,010,618
20110501	$4,027,080
20110701	$713,117
20110801	$3,356,069
20110901	$882,900
Total	**$80,450,597**

FactInternetSales

DateKey	SalesAmount
20101229	$14,477
20101230	$13,932
20101231	$15,012
20110101	$7,157
20110102	$15,012
20110103	$14,313
20110104	$7,856
Total	**$29,358,677**

DateKey	Reseller SalesAmount	Internet SalesAmount
20101201	$489,329	
20101229		$14,477
20101230		$13,932
20101231		$15,012
20110101	$1,538,408	$7,157
20110102		$15,012
20110103		$14,313
20110104		$7,856
20110105		$7,856
20110106		$20,910
20110107		$10,557
20110108		$14,313
20110109		$14,135
20110110		$7,157
20110111		$25,048
20110112		$11,231
20110113		$14,313
20110114		$14,135
Total	**$80,450,597**	**$29,358,677**

Figure 5-9. Chasm Trap is not a problem in Power BI

- Similarly, Power BI is fail-safe against the *Fan trap* problem. In Figure 5-10 you see the `Freight` per day (stored in the `SalesOrderHeader` table) and the `OrderQty` per day (stored in the `SalesOrderDetail` table, which has a many-to-one relationship to the `SalesOrderHeader` table). In the table visual on the right you see, that the `Freight` is not wrongly duplicated per day, but shows the same values as in the `SalesOrderHeader` visual.

SalesOrderHeader

DateKey	Freight
20110531	$15,051
20110601	$348
20110602	$375
20110603	$179
20110604	$375
20110605	$358
20110606	$196
Total	**$3,183,430**

SalesOrderDetail

DateKey	OrderQty
20110531	825
20110601	4
20110602	5
20110603	2
20110604	5
20110605	4
20110606	3
Total	**274914**

DateKey	Freight	OrderQty
20110531	$15,051	825
20110601	$348	4
20110602	$375	5
20110603	$179	2
20110604	$375	5
20110605	$358	4
20110606	$196	3
20110607	$196	3
20110608	$523	6
20110609	$264	3
20110610	$358	4
20110611	$353	4
20110612	$179	2
20110613	$626	7
20110614	$281	4
20110615	$358	4
20110616	$353	4
20110617	$174	2
Total	**$3,183,430**	**274914**

Figure 5-10. Fan Trap is not a problem in Power BI

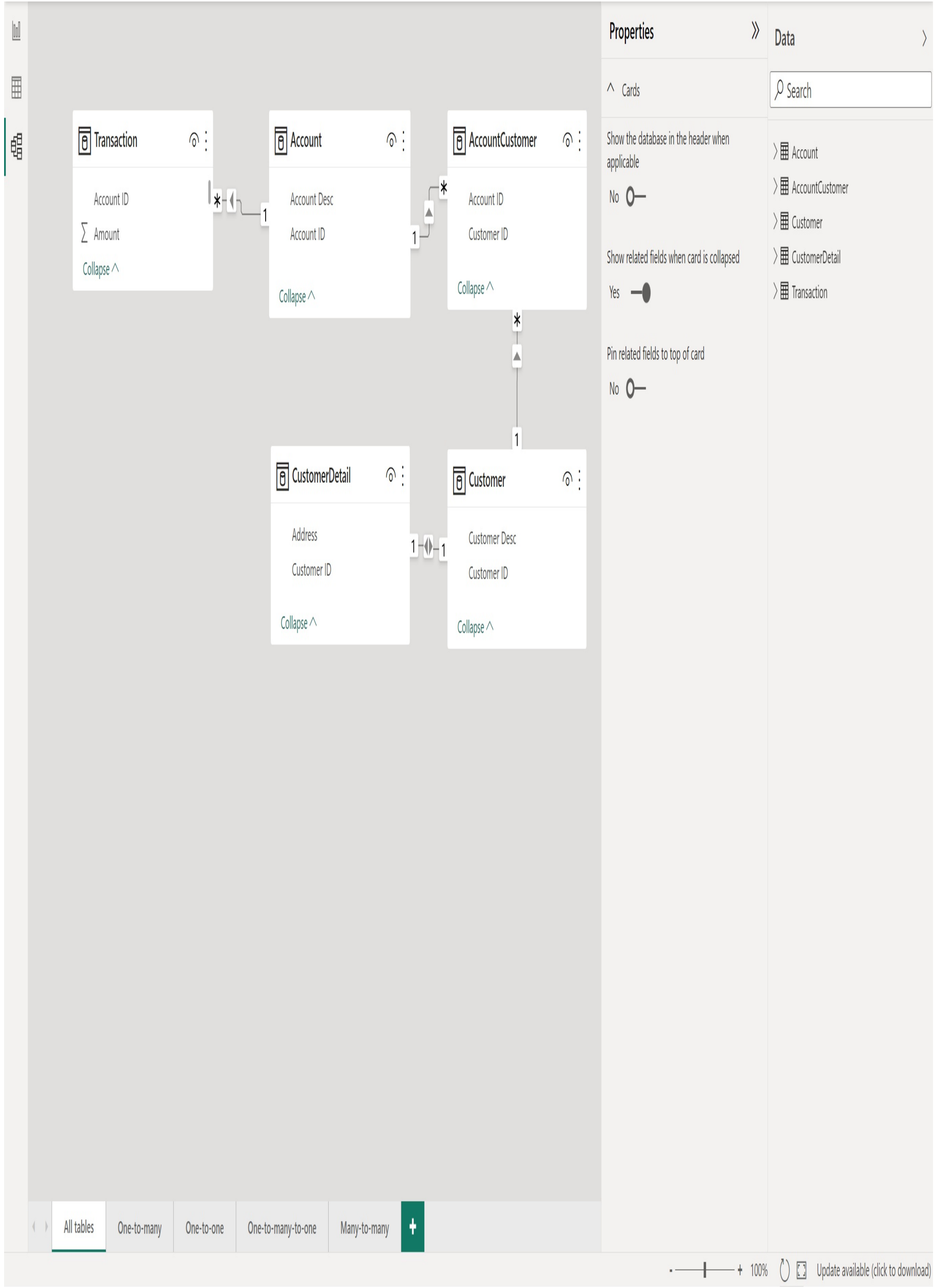
A good way to document the relationship, and therefore the possible join paths, is to show the data model as an Entity-Relationship Diagram, as you will learn in the next section.

Entity-Relationship Diagrams (ERD)

The model view in Power BI (and the diagram view for Analysis Services projects in Visual Studio) are exactly what you would draw in an Entity-Relationship diagram if you want to document the tables and their relations. In the model view you see "1" for the one-side and "*" to represent the many-side of a relation.

As the model view is not about foreign keys, but about filters, it additionally shows the direction of a filter, which can go either into one single direction or into both directions, represented by a small triangle (for single-directed filters) and two small triangles (for bi-directional filters).

In Figure 5-11 you can see a single-directed, many-to-one relationship between `AccountCustomer` and `Customer` and a bi-directional, one-to-one relationship between `Customer` and `CustomerDetail` shown in Power BI's model view.



Transaction
Account ID
Amount
Collapse
Account
Account Desc
Account ID
Collapse
AccountCustomer
Account ID
Customer ID
Collapse
CustomerDetail
Address
Customer ID
Collapse
Customer
Customer Desc
Customer ID
Collapse
Properties
Cards
Show the database in the header when applicable
No
Show related fields when card is collapsed
Yes
Pin related fields to top of card
No
Data
Search
Account
AccountCustomer
Customer
CustomerDetail
Transaction
All tables
One-to-many
One-to-one
One-to-many-to-one
Many-to-many
100%
Update available (click to download)

Figure 5-11. Power BI’s Model view

Data Modeling Options

Types of Tables

There is no explicit property for a table to indicate the usage of the table (e. g. entity table, lookup table, bridge table). I also have the strong opinion that the type of a (non-hidden) table should not be indicated by hints in its name (e. g. *Fact_Sales* or *Dim_Customer*). I instead recommend, that the names should be user-friendly for the report creators (who usually don’t care about what role a table plays in the data model, as long its returning the expected results).

The role of a table is just given by its relation to other tables. Fact tables are always on the many-side of a filter relationship. Dimension tables are always on the one-side in a star schema. In a snowflake schema they might as well be on the many-side in a relation to another dimension table. For example, the `Product` table will be on the one-side of the relationship to the `Sales` table (which is on the many-side, as each order line contains one single product, but the same product will be ordered several times over time). The `Product` table will be on the many-side in a relationship to the `Product Subcategory` table, as many products might share the same subcategory.

In Figure 5-8 you already saw a many-to-many relationship between tables `Budget` and `Product`. The reason, why this relationship has a many-to-

many cardinality is because the budget was not created on the `Product`'s table granularity, but per `Product Group` instead. The `Budget`'s table foreign key `Product Group` is not referencing the `Product` tables primary key (`Product Key`). In table `Product` column `Product Group` is not unique, the same `Product Group` will be found in several rows. As the join key is not unique in either table, Power BI restricts a direct relationship to cardinality many-to-many.

Relationships of cardinality many-to-many have some disadvantages in Power BI, as laid out in "Cardinality". A bridge table resolves a many-to-many relationship and is put between two tables, which are logically connected by a many-to-many relationship. The bridge table is replacing a many-to-many relationship with two one-to-many relationships. It is always on the one-side of the two relationships. The content of the bridge table is a distinct list of key(s) used to join the two original tables. As the content is only relevant for creating the relationship, but not for building a report, the bridge table should be hidden from the user. I usually put the postfix "BRIDGE" in the name of a bridge table. It makes it easier for me to spot the bridge tables, and therefore many-to-many relationships in my data model.

Figure 5-12 shows the model view of three tables of different type. The table on the very left is a fact table (`Budget`) and is on the many-side of the relationship. Right of this table you see a "bridge" table (`Product Group BRIDGE`), which bridges the many-to-many relationship between the `Budget` and `Product` table. The "bridge" table is on the one-side of both

relationships. The table on the very left is a dimension table (`Product`). It is on the many-side of the relationship, as the `Budget` table is not referencing the `Product` 's table primary key (`Product Desc`), but the non-unique column `Product Group` . You will learn more about the specifics of this data model in [Link to Come]>. Here I just used it to demonstrate different table types.

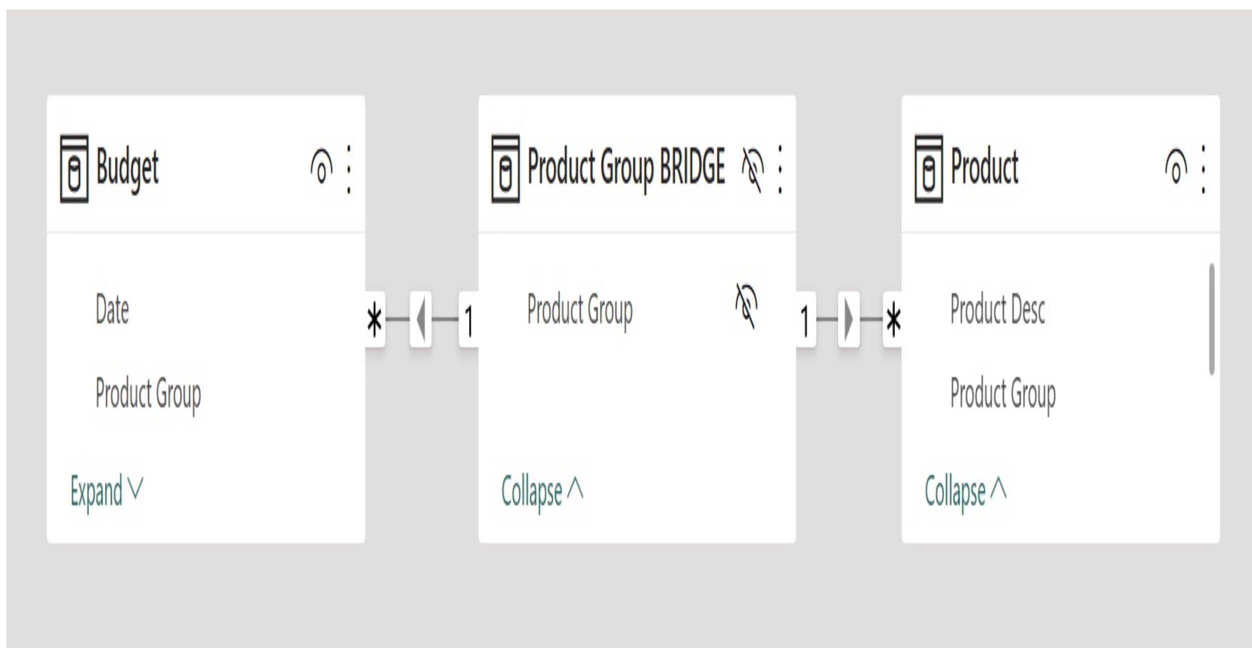


Figure 5-12. Tables of different types

Maybe you think, why should you bother with different (kind of) tables when you can just just store everything into one singe table? In the next section you will learn, why this is a bad idea, when it comes to Power BI and Analysis Services Tabular.

A Single Table To Store It All

While putting all information into a single table is common in many use cases or tools, and even Power BI allows you to build such a model, I would strongly discourage you from doing that. If you think in terms of a star schema, a single table means that the information of all dimension tables is stored inside the fact table. There are plenty of reasons, why you should split the information in at least two separate tables:

Size of model

Because Power BI's storage engine stores imported data in memory, Microsoft made sure to compress the data. The compression algorithm works very well for a star schema, but replicated dimensional information in every single row of a fact table does not compress very good. In a scenario I built to try this out, the single table used almost three times the space of a star schema. That means, that you can only store a third of the data in a single table, compared to a star schema on a given infrastructure. And the more memory the storage engine has to scan, the longer it will take and the more pressure is on your CPU as well. Transforming the single table into a star schema will help to fully benefit from the storage engine. The model size will be smaller, the reports will be faster.

Adding new information may be difficult

If you want to extend the data model later by adding additional information, you would either need to implement a transformation to add the data to the existing table – which can be dreadful (by aligning the different granularities of the existing and the new information) and you

would just increase the problems you are already facing with the single table. Or you would add the new information as a separated table. This would only work, if you need to join the two tables on one single dimensional column, as you can't have more than one active relationship. Joining two fact table directly is not recommended due to the size of the tables. Transforming the single table into a star schema will make it easier to extend the model. You would just add new dimension tables, re-use the already existing dimension tables and connect new fact tables to the existing dimensions.

Wrong calculations

I guess, everybody wants to make sure, that the numbers reported are the correct ones. Due to some optimization in the storage engine, queries on a single table might though result to wrong results, as laid out in the following example. Table 5-1 shows a simple and small table, containing `Date`, `ProductID`, `Price`, and `Quantity`.

Table 5-1. A simple table containing some sales

Date	ProductID	Price	Quantity
2023-02-01	100	10	3
2023-02-01	110	20	1

2023-02-02	110	30	4
2023-03-03	120	100	5

I then created three measures: One to count all rows of the Sales table (`# Sales = COUNTROWS(Sales)`) and two others, where I assume, that I want to count the rows of the `Sales` table independently of any filter on the `Date` colum. One version removes all filters from the `Date [Date]` column and the other removes the filter from the `Sales[Date]` column (by applying function `REMOVEFILTERS()`).

Figure 5-13 shows the formula of [# Sales] and a card visual next to it, which shows the value of 1. This number is correct: there was only one single sale for the filtered date of February 2 2023.

Figure 5-13. A report showing the count of rows of the `Sales` table.

[Link to Come] shows the formula and the content of `[# Sales ALL(Date[Date])]` , which shows a value of 3. There is one slicer per dimension: `Date` (with the second of the month selected) and `Product` (with `ProductID` 100 and 110 selected). Measure `[# Sales ALL(Date[Date])]` calculates the expected value of 3, because, if we

remove the filter on the `Date` (for the second of the month) we are left with only a filter on `ProductID`. For the two selected products (100 and 110) there are three rows available in the `Sales` table.

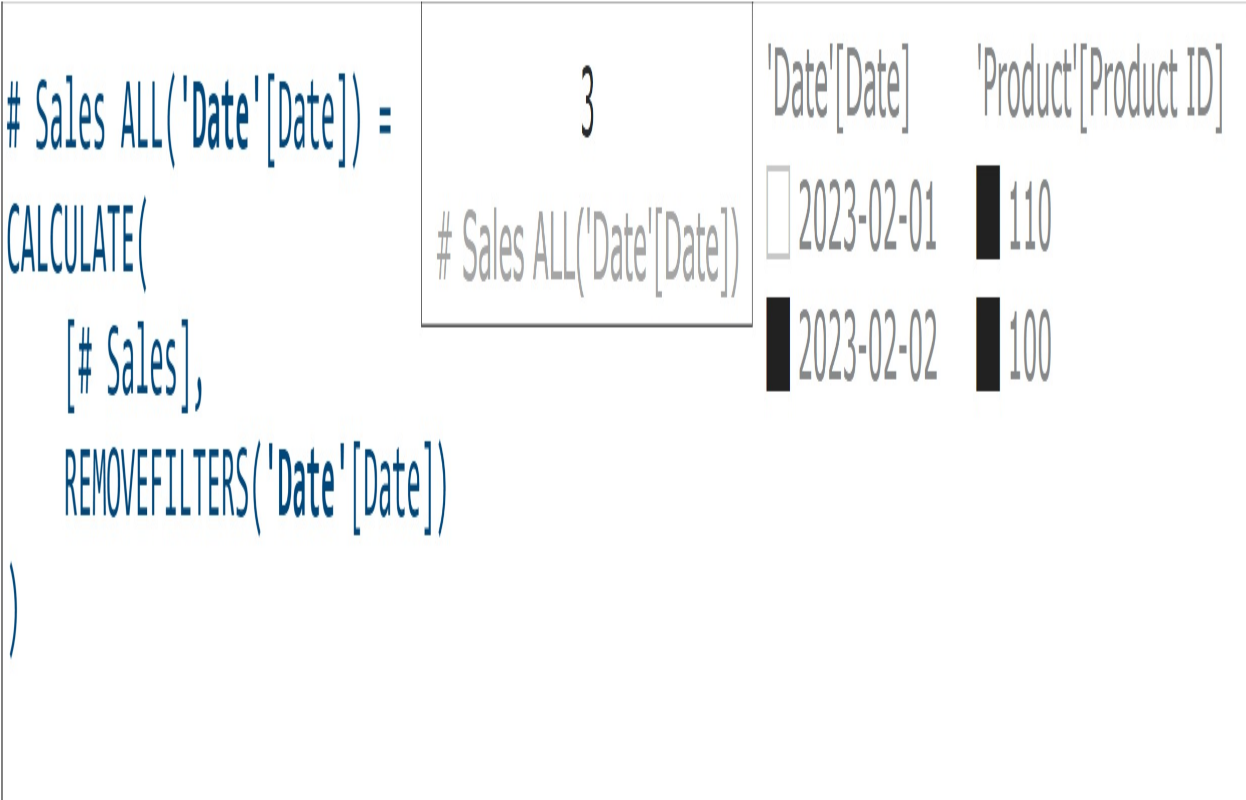


Figure 5-14. A report showing the count of rows of the `Sales` table for all dates.

The third section in the report shows a very similar content, but for measure `[# Sales ALL(`*`Sales`*`[Date])]` and filters on two columns of the `Sales` table (`Date` and `Product ID`) with the identical selection as on the dimensions. Unfortunately `[# Sales ALL(`*`Sales`*`[Date])]` shows an unexpected value of 2. Removing the filter from the *Sales*[Date] column should lead to a result of three.

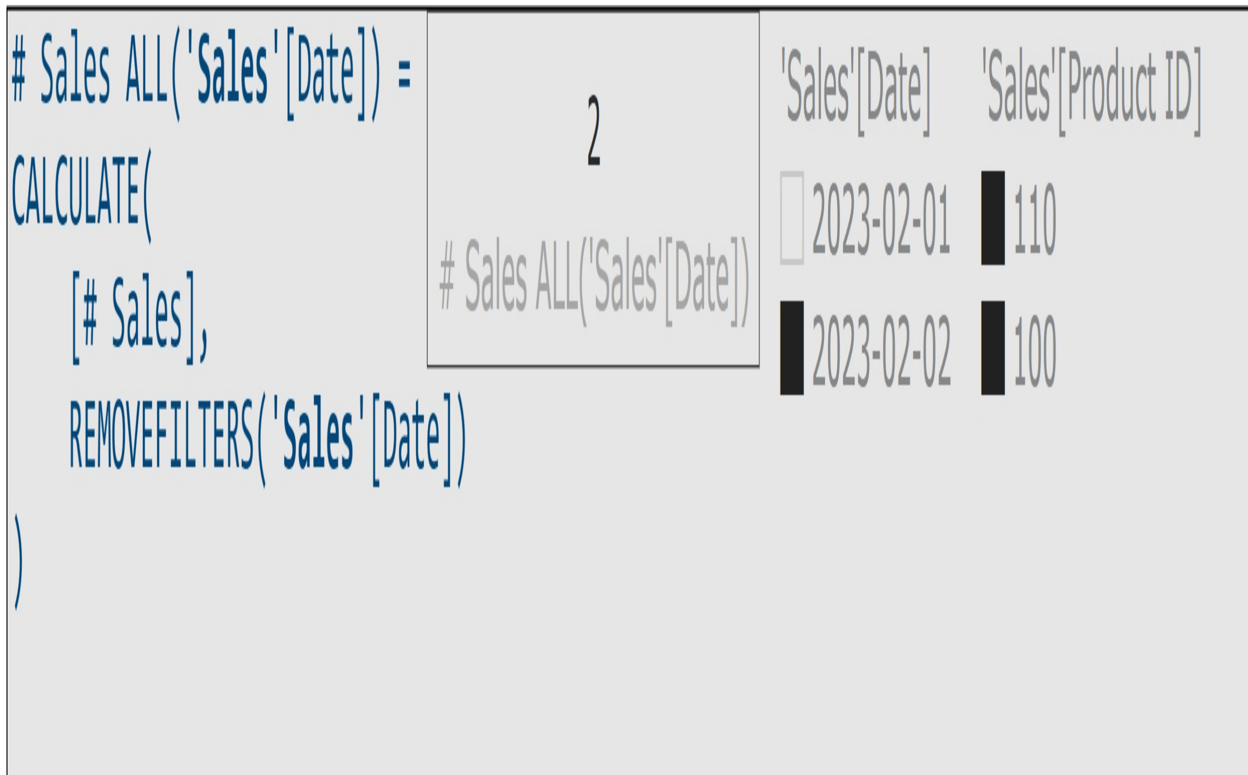


Figure 5-15. A report showing the wrong count of rows of the `Sales` table for all dates.

The reason why this measure shows an unexpected value is due to an optimization in the storage engine of Power BI / Analysis Services Tabular. At the point in time, when the `REMOVEFILTERS()` function kicks in, the `Sales` rows with `ProductID` of 100 were already filtered out to speed up the queries (as there are no `Sales` for this `ProductID` on the second of the month), leaving only rows for `ProductID` 110. Removing the filter on the `Sales[Date]` column does not recover the rows for `ProductID` 100. That’s why this measure only counts the rows for `ProductID` 110. This effect is not limited to counting, but affects all kind of measures, as soon as you start manipulating the filter context, which is very common even for rather simple calculations. This optimization (and the negative effect) only

happens when filters directly on the fact table are in place; filters via dimension table are not affected by this optimization.

That was the long explanation why you should not create a single table data model in Power BI or Analysis Services Tabular, if you care about performance, an easy-to-understand data model and correct numbers. The short version is: Always create a dimensional model, never put all columns into a single table. And never filter directly on the fact table (but hide those columns from the users).

And don't forget: If you now start with the single table, and first later discover that you want to change it to a star schema, this step will break all your existing reports based on this model.

On the other extreme of a data model, you do not put everything into one single table, but fully normalize the data model to avoid redundancy. This is not such a great idea for a data model inside of Power BI, as you will learn in the next section.

Normal Forms

Power BI is rather flexible about what kind of data models it allows you to build. Unfortunately, a normalized data model comes with a lot of tables and relationships. Such a model will be hard to understand for the report users (as information even from the same entity is spread out over multiple tables).

And all the different tables need to be joined, according to their filter relationship. Joins are expensive, leading to slower reports. Normalized data models are optimal for application databases, but not for analytical databases.

Again: If you now start with a normalized data model, and first later discover that you want to change it to a star schema, this step will break all your existing reports based on this model. Better start with a dimensional model up front. The next section will remind you once again.

Dimensional Modelling

You do not need to believe that dimensional data modelling is so much better than all the other data modelling approaches. But, please, trust me, that Power BI and Analysis Services Tabular (to be more precise: their storage engine, called VertiPaq) is optimized for dimensional data models through all its fibers. That’s why it is the goal not to just load the data as it is into Power BI, but to transform the tables you get from the data source into a dimensional data model.

There is no need to actually store the data already in a dimensional model in the data source. For example, Bill Inmon (mentioned in chapter 1), recommends to store all analytical data in a fully normalized schema (which he calls the *Corporate Information Factory*). Only the data mart layer is a dimensional model. This layer on the other side, can either be derived from the normalized schema with the help of DAX, Power Query or SQL. In this

book I will teach you all necessary steps in all three languages – so no excuses anymore!

An important question you have to ask the report users (or yourself): How much of detail is really necessary in the reports? Does the report need to cover every single transaction or are aggregates enough? Is it necessary to store the time portion, or will the report only be generated on a month basis? The answers to these question define the *Granularity* of the data model. Read on, to learn more.

Granularity

It is important that the granularity of your fact table matches the primary keys of your dimension tables, so you can create a filter relationship between them with a one-to-many cardinality. In chapter 6 in the section “Budget” you will see an example for a case, where new information needs to be added to an existing data model (the budget) which has a different level of detail: The budget is only available per product group, but not per product. The actual sales data, on the other hand, is on the product level. The solution is to add the budget as a fact table on its own. (Chapter 6 will also explain how you can create a filter relationship between the product table and the budget table, despite the different granularity).

No matter which kind of data source you need to do analytics on, the shape of it will most probably not directly fit into a dimensional model. Luckily, we

have tools available to extract, transform and first then load the data into Power BI. The next section got you covered on these challenges.

Extract, Transform, Load

The process to extract, transform, and load the data (short: ETL), is not done via the model view, but you can use either DAX, Power Query / M, or SQL to achieve this. Beginning with chapter 10, you will dive into those languages and make them your tool to extract and transform the data as needed.

Read on to learn, that there is a special kind of transformation necessary to implement *Slowly Changing Dimensions*, which is not done in the *Model view*.

Key Takeaways

In this chapter I took the basic concepts of data modeling and matched it with features available in Power BI and Analysis Services Tabular. You learned a lot about the *Model View*:

- The *Model view* (in Power BI) and the *Diagram view* (in Visual Studio) gives you a graphical representation of the tables and their relationship in the data model and allow you to set plenty of properties, for both the tables and their columns.
- The purpose of the filter relationship is to propagate filters from one table

to another. The propagation works only via one single columns and is implemented as an equi-join.

- The filter relationship between two tables is represented by a continuous line (for active relationships) or a dashed line (for inactive relationship). The latter can be activated in a DAX measure, which will be discussed in chapter 10.
- The cardinality of a relationship is represented by “1” or “*” (= many). You can create filter relationships of type one-to-many (which is the most common), one-to-one and many-to-many. Many-to-many relationships can be created unintentionally when both columns contain duplicates by mistake. One-to-one relationships can be created unintentionally when both columns contain only unique values. Double-check those cases.
- A filter relationship has a direction. A filter can be either propagated from one table to another or in both directions. Bi-directional filters bear the risk of making a data model slow. There can be situations where you cannot add another table with a bi-directional filter, when it would lead to an ambiguous model.

You now have an understanding of how important a data model in Power BI is. The next chapter will teach you practical knowledge of how to shape it into a dimensional model.

Chapter 6. Building a Data Model in DAX

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sevans@oreilly.com*.

With DAX you are writing *Data Analytic Expressions* – which allow you to create calculated tables, calculated columns and, most important, measures (and row level security and queries, which are not in the scope of this book). All you can achieve with calculated tables and columns you can also achieve with solutions in Power Query / M and with SQL. If you are already familiar with either Power Query or SQL, you might first try those to transform your data model. If you just started with Power BI, then you need to learn DAX anyways, as some problem can only be solved with measures, written in

DAX – and you might also implement the transformation to build your data model in DAX as well.

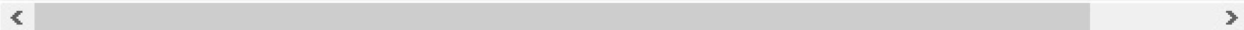
TIP

By looking at a piece of DAX code it can sometimes be hard to recognize if it is the definition of a calculated column, a measure or a calculated table. Therefore, I use the following convention:

```
[Measure Name] *:=* _<definition of a measure>_

**'Table Name'**[Column Name] *=*
   <definition of a calculated column>

[Table Name] **= /* calculated table */** _<definition of a calculated ta
```



Normalizing

As you learned back in chapters 1 and 2, normalizing is important for your fact tables, and means, that you strip the table from replicating information. You only keep foreign keys to one or more different tables, which contains `DISTINCT` lists of the otherwise redundant information. These other table are the dimension tables.

With that said, normalizing is as easy as just removing all column with repeating information, which are not the (primary) key of the information and putting them into a different table of its own. To find out, which columns

contain repeating information, I just create a table visual in Power BI with only the single column or a combination of columns which might have a one-to-one relationship with each other, which you can see in Figure 6-1. Power BI will automatically only show the distinct values. In the given example, the following combinations of columns are candidates for building dimensions:

- Country
- Discount Band
- Product, Manufacturing Price
- Date, Month Name, Month Number, Year
- Segment

Discount Band
High
Low
Medium
None

Segment
Channel Partners
Enterprise
Government
Midmarket
Small Business

Country
United States of America
Mexico
Germany
France
Canada

Product	Manufacturing Price
Amarilla	260
Carretera	3
Montana	5
Paseo	10
Velo	120
VTT	250

Date	Month Name	Month Number	Year
2013-09-01	September	9	2013
2013-10-01	October	10	2013
2013-11-01	November	11	2013
2013-12-01	December	12	2013
2014-01-01	January	1	2014
2014-02-01	February	2	2014
2014-03-01	March	3	2014
2014-04-01	April	4	2014
2014-05-01	May	5	2014

Figure 6-1. Dimension candidates

Always also use your domain knowledge and discuss with the domain experts to decide, if choosing these candidates are indeed a good decision. Especially if you only work with demo or test data (and not productive data), the true relationship between the columns might not be clear from just looking at the data available.

When we can agree on, that the candidates above are the true dimensions, you can create a calculated table and use `DISTINCT` with either a single column reference (which will work for `Country`, `Discount Band` and `Segment` in our example) or in combination with `SELECTCOLUMNS` (for columns `Product` and `Manufacturing Price`). Maybe you ask yourself now: Is it really worth to create single-column dimensions? The clear answer is: Yes! Remember, when we talked about all the disadvantages and problems you get with a single-table model back in chapter 5? We must avoid direct filters on the fact table in all circumstances.

You should overcome the temptation to create the `Date` table in the same manner (by applying `DISTINCT` over the fact table's columns), as the date values in the fact table usually have gaps (e. g. weekend, bank holiday, etc.), which you can clearly see in Figure 6-1. I will show you a way of how to create a full-functional date table later in section "Time and Date".

We can though not physically remove any of the columns above from the fact

table, but only hide those columns (so that the report creators are not unintentionally using them) – otherwise the creation of the calculated tables (referencing those columns) would fail.

WARNING

And that's the big disadvantage of using DAX to model your data: You can only add columns and tables to form a star schema, but you still need to keep all the information in it's original (un-modeled, non-star) shape.

You cannot truly transform the data into the intended model. But this should not keep your from applying these best practices. It is better that you shape your data with DAX then not shaping it at all. Just remember, that when the size of the data model (in memory – but you can also just take a look on the size of the PIBIX file on disk to get a rough impression how small or big your data model is) is starting to become a problem for your resources, then it's time to re-factor your DAX into Power Query (or SQL).

Denormalizing

I am sure, by now, you are already familiar, that we need to denormalize the dimension tables. To denormalize, we need to add columns with information from a related table into the main dimension table. The DAX function which achieves this is called RELATED. It can traverse from the many side of a relationship to the one side, even over several tables, and fetch the content of

a column. In the example, a product's information is split into three different tables:+DimProduct+, `DimProductSubcategory`, and+DimProductCategory+. Simply create two new calculated columns in table `DimProduct`:

```
DimProduct[Subcategory] = RELATED(DimProductSubcategor
DimProduct[Category] = RELATED(DimProductCategory[Engl
```

As there is a direct relationship in the data model between DimProduct and `DimProductSubcategory`, it makes sense, that we can reference a value from there. But DAX is smart enough to traverse also from DimProduct over `DimProductSubcategory` to `DimProductCategory`. Therefore, the second example works as expected. Be reminded, that RELATED can only reference from a table on the many side to a table on the one side. (To traverse the other direction, you can use RELATEDTABLE, which returns a table with all the values from the many side.) Again, we can (and definitely should) hide the two tables DimProductSubcategory and `DimProductCategory`, to avoid that report creators will use any of the columns unintentionally, but we cannot actually delete the two tables from the model (because, then the newly created calculated columns would through an error).

Calculations

Calculations is the home game for DAX. DAX stands for Data Analysis Expressions and is therefore built for creating formulas for even very complex challenges. And when I speak of calculations I mostly mean (explicit) measures, that’s what the core competence of a Data Analytic Expression is. Creating calculated tables and calculated columns is possible as well, but I consider them more as a workaround in the early days of Excel’s Power Pivot, when Power Query was not available yet. In many scenarios you are better off with explicit measures.

You should consider explicitly creating measure for all type of calculations:

- Simple aggregations for additive calculations, which could also be calculated through `Default Summarization`. In the general part about Power BI I already laid out why you should explicitly create DAX measures instead of relying on `Default Summarization`. I usually rename the numeric column (e. g. add an underscore “+” as a prefix), hide the column and then create a simple measure by applying the SUM function (or whatever aggregation is making sense). When the calculation is more complex (e. g. because you need to multiply the quantity with a price) you need the SUMX function (or a comparable iterator function), where you can provide the formula for the multiplication. SUMX is calculating this formula for each and every row of the table you provided as the first parameter of the function and sums these results up.

  ```
  [Units Sold] :=
  ```

```
    SUM(Financials[+Units Sold])

[Gross Sales] :=
    SUMX(
        'Financials',
        'Financials'[+Units Sold] * Financials[+Sale
    )
```

- Semi-additive calculations require you to specify for which date the value should be calculated. Usually, it is the first or the last date of the current time range.

```
[First Value] :=
/* based on a blog post by Alberto Ferrari
   https://www.sqlbi.com/articles/semi-additive-meas
 */
VAR FirstDatesPerProduct =
    ADDCOLUMNS (
        VALUES ( 'Product'[Product ID] ),
        "MyDay", CALCULATE ( MIN ( 'Sales'[Date] ) )
        )
    )
VAR FirstDatesPerProductApplied =
    TREATAS (
        FirstDatesPerProduct,
        'Product'[Product ID],
        'Date'[Date]
```

```
    )
VAR Result =
    CALCULATE (
        SUM ( 'Sales'[Quantity] ),
        FirstDatesPerProductApplied
    )
RETURN Result

[Last Value] :=
/* based on a blog post by Alberto Ferrari
   https://www.sqlbi.com/articles/semi-additive-meas
 */
VAR LastDateInContext = MAX ( 'Date'[Date] )
VAR LastDatesPerProduct =
    ADDCOLUMNS (
        CALCULATETABLE (
            VALUES ( 'Product'[Product ID] ),
            ALL ( 'Date' )
        ),
        "MyDate", CALCULATE (
            MAX ( 'Sales'[Date] ),
            ALL ( 'Date' ),
            'Date'[Date] <= LastDateInContext
        )
    )
VAR LastDatesPerProductApplied =
    TREATAS (
        LastDatesPerProduct,
```

```
            'Product'[Product ID],
            'Date'[Date]
        )
    VAR Result =
        CALCULATE (
            SUM ( 'Sales'[Quantity] ),
            LastDatesPerProductApplied
        )
    RETURN Result
```

TIP

Non-additive calculations must be done in the form of a DAX measure. You cannot achieve the correct results with any other technique (e. g. calculated column, Power Query, SQL, etc.)

- Results of non-additive calculations cannot just be aggregated in a meaningful sense. Therefore, you need re-create the calculation as a DAX measure based on the aggregated parts of the formula. You need to sum up the elements of the formula (instead of summing up the results). The `Margin in Percentage of the Sales` is calculated by dividing the margin by the sales amount, which works perfectly on the level of one single row in the sales table. But a report barely shows the individual sales rows, but aggregated values. Calculating the sum or even the average of the result of the division would show the wrong value. Therefore, it needs to be calculated as shown.

```
[Margin %] := DIVIDE(SUM('Sales'[Margin]), SUM('Sale
```

This measure will work on the level of individual sales, where only a single sale event is available, as well (as the sum of the margin of a single row in the sales table is just the margin of the row).

Counts over `DISTINCT` entities are another example for non-additive calculations. The `DISTINCT` count of customers who bought something in the first quarter of a year is not the sum of the `DISTINCT` counts of customers in January plus the `DISTINCT` counts of customers in February plus the `DISTINCT` customers in March, as some customers might have bought something in more than one month. Those customers may not be counted twice when calculating the `DISTINCT` count for the quarter. But creating such a measure is not a big deal:

[[+DISTINCT+Count]]

```
[+DISTINCT+ Count of Products] := +DISTINCT+COUNT('Sal
```

[[CountVs+DISTINCT+Count]] .Visual showing measures `Count of Products` and `Distinct Count of Products`

image::ch10_BuildingADataModelInDAX/CountVs+DISTINCT+Count.png[

You can see in [Link to Come], that two products were sold on the first of the month (A and B), and a single product each on the second (B) and third ©. But in total it has been only three different products (A, B, and C), which

were sold during those three days (as product B was sold on both, the first and the second of the month). The column `Count of Products` adds up to 4 products, while `Distinct Count of Products` shows the correct total of 3 (different) products. Sometimes I see people complaining on social media, that the table visual in Power BI is buggy, as it does not always add up the individual numbers in the total. I don't really get these discussions, as it clearly depends on the context of a calculation, if the individual numbers need to be aggregated or if the calculation has to be done on the aggregated level, as pointed out in the discussion of additive and non-additive measures back in the first part of this book.

TIP

Non-additive calculations must be done in the form of a DAX measure. You cannot achieve the correct results with any other technique (e. g. calculated column, Power Query, SQL, etc.)

- Time Intelligence calculations are another use case, which can only be solved with DAX measures. The trick is basically to use CALCULATE to change the time period accordingly (e. g. from the current day to all days since the beginning of the year to calculate the year-to-date value), similar to the logic for the semi-additive measures. DAX comes with built-in functions to either directly calculate the value (e. g. TOTALYTD) or functions which you can use as a filter parameter for CALCULATE (e. g. DATESYTD). Those functions are just hiding some complexity from you, but you can always come up with a formula which is achieving the same

result (even with the same performance) by e. g. calculating the first day of the year and then changing the filter context accordingly. See three implementations of a year-to-date calculation for `Sales Amount` in the following code snippets:

```
[TOTALYTD Sales Amount] :=
TOTALYTD(
    [Sales Amount],
    'Date'[Date]
)

[TOTALYTD Sales Amount 2] :=
CALCULATE(
    [Sales Amount],
    DATESYTD('Date'[Date])
)

[TOTALYTD Sales Amount 3] :=
CALCULATE(
    [Sales Amount],
    DATESBETWEEN(
        'Date'[Date],
        STARTOFYEAR(LASTDATE('Date'[Date])),
        LASTDATE('Date'[Date])
    )
)
```

All three have the same semantic and their different syntax generates the identical execution plan. Therefore, their performance is identical. They are just using more or less syntax sugar to write the code.

Figure 6-2 shows the identical result of the three different approaches in DAX to calculate the YTD.

Sales

Date	Product	Price	Quantity
01.09.2021	A	10	3
01.09.2021	B	20	1
02.09.2021	B	30	4
03.09.2021	C	100	5

Date	Sales Amount	TOTALYTD	TOTALYTD 2	TOTALYTD 3
01.09.2021	50	50	50	50
02.09.2021	120	170	170	170
03.09.2021	500	670	670	670
04.09.2021		670	670	670
Total	670			

Figure 6-2. Visual showing the result of the three different measures to calculate the year-to-date value for `Sales Amount`

TIP

Time Intelligence calculations must be done in the form of a DAX measure. You cannot achieve the correct results with any other technique (e. g. calculated column, Power Query, SQL, etc.)

Especially requirements for time intelligence can easily lead to many

variations of a single measure (e. g. year-to-date, previous month, previous year, differences in absolute numbers, differences in percentage, comparison to budget, etc.). It can be very tedious to create (and maintain) all the variations for each measure. Here, *Calculation Groups* come in very handy. *Calculation Groups* add a layer above all measures and are explicitly activated as filters within visuals or via `CALCULATE` within other measures. The advantage is, that you only need to specify the logic of how to calculate e. g. a year-to-date for a measure as one single item in the calculation group. When you create a calculation item, you can simply copy and paste an existing definition of a measure, but replace the base measures name (e. g. `[Sales Amount]`) with function `SELECTEDMEASURE`. This logic can then be activated for every measure when you need it. If the logic changes, you need only change it in a single place (the calculation item), instead of changing it per measure. *Calculation Groups* are fully supported in Power BI - but at time of writing the user interface of Power BI Desktop does not expose their definitions. Therefore, you need to use a third-party tool to create and maintain *Calculation Groups* in your PBIX file. If you are working with Analysis Services Tabular you have full access to the definition of *Calculation Groups* in, for example, Visual Studio.

```
[Actual] := SELECTEDMEASURE()

[YTD] := TOTALYTD(SELECTEDMEASURE(), 'Date'[Date])
```

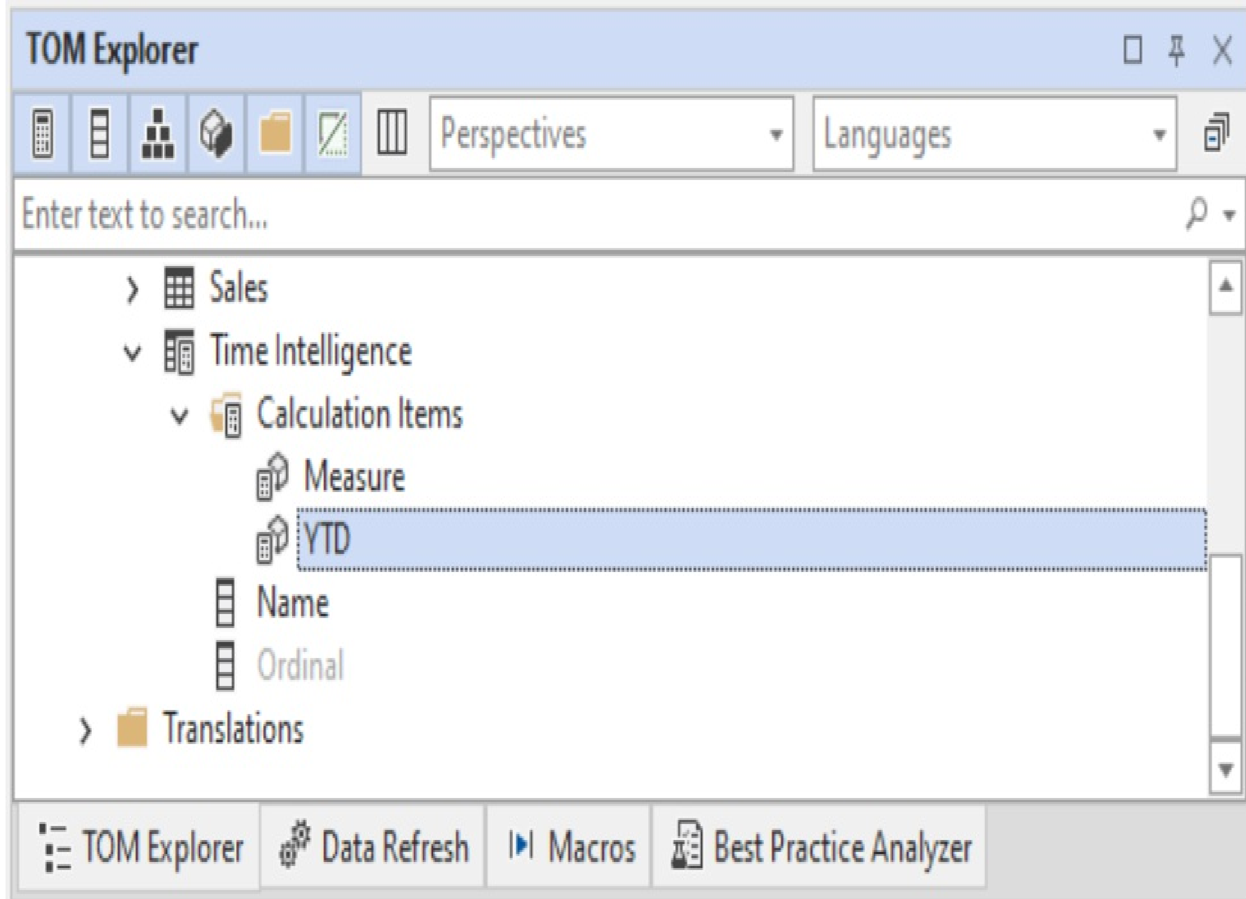


Figure 6-3. Defining a *Calculation Group* in Tabular Editor 3

In the screenshot I used Tabular Editor 3, but you can also use the free version of Tabular Editor (Version 2) to maintain *Calculation Groups*. In the first step you need to create a new *Calculation Group* by right-clicking `Tables` inside `TOM Explorer`. I renamed both, the table and column “Name” to “Time Intelligence”. Second, you add `Calculation Items` per variance. Here I added one for `Actual` and one for `YTD` as just described.

Flags and Indicators

Replacing abbreviations and technical identifiers with meaningful text can easily be achieved with DAX. I intentionally used a different syntax for each of the following examples, to demonstrate different possibilities:

`IF` Function

Every replacement logic can be implemented by writing a bunch of nested IF functions. Always make sure to use a new line for each of the parameters and indent the parameters. Otherwise, a formula, especially one with nested functions can be really hard to read. If the first parameter of IF evaluates to TRUE, then the second parameter is returned. Otherwise, the third parameter. Calculated column `Class Description` shows three nested IF functions.

+

```
'DimProduct'[Class Description] =
IF(
    DimProduct[Class] = "H",
    "High",
    IF(
        DimProduct[Class] = "M",
        "Medium",
        IF (
            DimProduct[Class] = "L",
            "Low",
            "other"
        )
```

```
    )
)
```

`SWITCH` Function

SWITCH can be used with just a simple list of values, which I prefer over nested IFs due to better readability. For calculated column `Product Line Description` I did provide a column name as the first parameter and different literals for the second, fourth, etc. parameter. If the first parameter matches one of the literals, then the third or fifth or etc. value is returned. I provide a last parameter in case there is a new `ProductLine` added after I wrote the formula. If I would omit the last parameter, then for such a new `ProductLine` a blank value would be shown as the `Product Line Description`. I prefer "other" (or something similar) over a blank text.

+

```
'DimProduct'[Product Line Description] =
SWITCH(
    DimProduct[ProductLine],
        "R", "Road",
        "M", "Mountain",
        "T", "Touring",
        "S", "Standard",
        "other"
)
```

`SWITCH TRUE` Function

`Finished Good Description` works with the SWITCH function, but in a different way. For the first parameter I used TRUE and the even parameters contain each a condition (which evaluates to TRUE or not) instead of a literal value. If the first parameter and the second parameter are equal (that means, that the condition provided in the second parameter evaluates to TRUE) then the third parameter is returned. If that's not the case, then the first parameter is compared with the fourth parameter and so forth. You should provide a last parameter which is returned when all the comparisons failed.

+

```
'DimProduct'[Finished Goods Description] =
SWITCH(
    TRUE(),
    DimProduct[FinishedGoodsFlag] = 0, "not salable",
    DimProduct[FinishedGoodsFlag] = 1, "salable",
    "unknown"
)
```

Lookup Table

Generally, I prefer to have a lookup table for the replacement values. I find it

easier to just maintain the content of a table instead of rewriting a formula when new values need to be added or existing replacements have to be updated. If you need the replacements in more than one language, than a lookup has its benefit as well (as we will discuss in chapter 3 when we talk about multi-lingual reports). Creating the lookup-table in DAX is clearly not my favorite (because changing the content of the table means to change the formula of the calculated table), but it can be done with the `DATATABLE` function. Table `Styles (DAX)` contains the code. When you then create a filter relationship between the `Styles (DAX)` and `DimProduct` over each column `Style`, you can use `RELATED` to lookup the values. In case a `Style` is present in `DimProduct` which is not (yet) available in table `Styles (DAX)` I check for `BLANK` and return "unknown".

+

```
[Styles (DAX)] = /* calculated table */
DATATABLE(
    "Style", STRING,
    "Style Description", STRING,
    {
        {"W", "Womens"},
        {"M", "Mens"},
        {"U", "Universal"}
    }
)
```

+

```
'DimProduct'[Style Description] =
VAR StyleDescription = RELATED('Styles (DAX)'[Style De
VAR Result =
IF(
    ISBLANK(StyleDescription),
    "unknown",
    StyleDescription
)
RETURN Result
```

Treating BLANK values

Sometimes you do not need to develop complex transformation, but only make sure to replace empty strings. DAX distinguishes two kinds of empty strings. A string can indeed contain just an empty string. This can be checked by comparing an expression against two double-quotes “”. Additionally, a string (or a column or expression of any data type) can also be blank. Blank means, that the string is not just an empty string, but that there was no value provided at all. Relational database call those missing values `NULL`. You can either compare an expression against `BLANK()` or you can explicitly check if an expression is blank by passing the expression into function `ISBLANK`. In calculated column `WeightUnitMeasureCode` I replaced empty and blank values with “N/A”.

+

```
'DimProduct'[WeightUnitMeasureCode cleaned] =
IF(
    ISBLANK(DimProduct[WeightUnitMeasureCode]) || DimP
    "N/A",
    DimProduct[WeightUnitMeasureCode]
)
```

Time and Date

As pointed out in chapter 6, you should create your own time-related table(s) when it comes to Power BI and Analysis Services Tabular. You can use the DAX code in this section as a template, which you then change and adopt to the needs of your report users. The number of rows in a `Date` or `Time` table is usually neglectable – so you do not have to limit yourself in the amount and variations of columns you want to add.

First, let’s create a `Date` table. The starting point is to create a list of dates for the time range your fact tables contain. Basically, you do have two options: `CALENDARAUTO` and `CALENDAR`.

> *`CALENDARAUTO`*
> Function `CALENDARAUTO` scans all your tables for columns of data type

`Date` and will then create a list of dates for January 1 of the earliest year until December 31 for the latest year. This will work as long as you do not import columns with “exotic” dates (like birthdates or placeholders like January 1 1900 or December 31 9999). In case of financial years (which do not start with January 1) you can pass in an optional parameter to `CALENDARAUTO` to move the start month by x months).

```
[Date (CALENDARAUTO)] = CALENDARAUTO() /* calculated
```

`CALENDAR`

Function `CALENDAR` gives you more control, as you have to provide two parameters: The first date and the last date of your `Date` table. These parameters can either be hardcoded (e. g. `DATE(2023, 01, 01)`), which is not very flexible and you need to set a reminder in your calendar to change the value once a year to add the dates for the new year) or you can write an expression where you calculate the two dates from your fact table’s date column. Unless your fact table is huge, the calculation will be fast enough and will give you rest-of-mind, that the date table will always contain all necessary entries with every refresh.

```
[Date (CALENDAR)] = /* calculated table */
    CALENDAR(
        DATE(
            YEAR(MIN('Fact Reseller Sales'[OrderDate
```

```
                01, /* January */
                01  /* 1st */
            ),
            DATE(
                YEAR(MX('Fact Reseller Sales'[OrderDate]
                12, /* December */
                31  /* 31st */
            )
        )
```

After you created the calculated table you can add new columns over the user interface of Power BI Desktop. I would though recommend to nest `CALENDARAUTO` or `CALENDAR` into `ADDCOLUMNS` and then specify pairs of name and expression for the additional columns. With that approach you have everything in one single place (the expression for the calculated table) and not spread out over separated calculated columns. This allows you also to easily copy and paste this full definition of the calculated table to the next data model.

```
[Date (CALENDAR)] = /* calculated table */
ADDCOLUMNS(
    CALENDAR(
        DATE(
            YEAR(MIN('Fact Reseller Sales'[OrderDate])
            01, /* January */
            01  /* 1st */
```

```
        ),
        DATE(
            YEAR(MX('Fact Reseller Sales'[OrderDate]))
            12, /* December */
            31  /* 31st */
        )
    ),
    "Year", YEAR([Date]),
    "MonthKey", YEAR([Date]) * 12 + MONTH([Date]),
    "Month Number", MONTH([Date]),
    "Month", FORMAT([Date], "MMMM"),
    "YYYY-MM", FORMAT([Date], "YYYY-MM"),
    "Weeknumber (ISO)", WEEKNUM([Date], 21)
    "Current Year", IF(YEAR([Date])=YEAR(TODAY()), "Cu

)
```

Typical additional columns for a date table are:

- `DateKey` as a whole number representing the date in the format `YYYYMMDD`. You can calculate this whole number by extracting the year from the date, which you multiple with 10000, add the number of the month multiplied by 100 and then add the day. In a data warehouse it is best practice to also have the keys for dates in the form of a whole number. In Power BI and Analysis Services Tabular this is not so important.

- `Year` as the year portion of the date. DAX function `YEAR` got you covered here.
- Variations of the month, like the month number of the year, the month name, the year and the month combined in different formats. Most of the variations can be calculated by using function `FORMAT` and passing in a format string.
- You pass the date as the first parameter for function WEEKNUM. The second parameter allows you to specify, if your week starts on Sundays or on Mondays or if the week number should be calculated according to the ISO standard.
- Users expect that a report shows the most recent data. Preselecting the right year and month can be a challenging task unless you have a column containing "Current Year" or "Current Month", which dissolves to the right year or month.

There are no functions similar to `CALENDARAUTO` or `CALENDAR` to get the range for a time table. But we can use `GENERATESERIES` to request a table containing a list of values for the specified range of integers. To create a table for every minute of the day, we need to CROSSJOIN a table containing values 0 to 23 (for the hours of a day) and a second table containing values 0 to 59 (representing the minutes of an hour).

Again, by using `ADDCOLUMNS` we can add additional columns to this expression, so we have the full definition of this calculated table in one single place: * Function `TIME` can convert the pairs of hours and minutes into a

proper column of datatype `Time` . * Function `FORMAT` can do its wonders also with time related content.

```
[Time (DAX)] = /* calculated table */
VAR Hours = SELECTCOLUMNS(GENERATESERIES(0, 23), "Hour
VAR Minutes = SELECTCOLUMNS(GENERATESERIES(0, 59), "Mi
VAR HoursMinutes = CROSSJOIN(Hours, Minutes)
RETURN
    ADDCOLUMNS(
        HoursMinutes,
        "Time", TIME([Hour], [Minute], 0),
        "Time Description", FORMAT(TIME([Hour], [Minut
    )
```

Role-Playing Dimensions

If you opt to load a role-playing dimension only once into the data model, you need to make sure, that you add as many relationships as foreign keys from one single table to the role playing dimension exist. A maximum of one of those relationships can be active, the others only can be inactive, but can be activated in measures. That means, that you need to create one variation of each measure per role. Instead of having just a `Quantity` measure, you would create individual measures like `Order Quantity`, `Sales Quantity`, etc. Each of these measure use `CALCULATE` and

`USERELATIONSHIP` to explicitly activate one of the relationships. DAX is smart enough to (implicitly) deactivate the active relationship for the sake of the context inside `CALCULATE`, so that still only one relationship is active at one point in time.

```
[Order Quantity] :=
CALCULATE(
    SUM(Sales[Quantity]),
    USERELATIONSHIP(Sales[OrderDate], 'Date'[Date])
)

[Ship Quantity] :=
CALCULATE(
    SUM(Sales[Quantity]),
    USERELATIONSHIP(Sales[ShipDate], 'Date'[Date])
)
```

TIP

Again, *Calculation Groups* can be of help here. You can create *Calculation Group* items per role of the dimension, instead of duplicating all your measures as many times as you have roles.

The alternative approach is to physically load the role playing dimensions several times. Instead of living with just one single `Date` table you will create calculated tables in DAX to duplicate the table (with all its content). This has the disadvantage of increasing the size of your model, but as long as

the size of the role-playing dimension is not huge, this is usually neglectable. The advantage is, that you do not need to create variations of your measures (by applying CALCULATE and USERELATIONSHIP), but the report creater choses one copy of the dimension table over the other – or can even combine both. Creating a copy of a table in DAX is rather easy. You just create a calculated table and use solely the name of the other table as the expression. I, though, strongly recommend renaming all columns to add e. g. the table name as the prefix, so it is clear which e. g. `Date` column is referred to (the one from the newly created `Order Date` or the `Sales Date`). You can do this by either manually renaming all columns or by changing the expression of just referring to the base table and use `SELECTCOLUMNS` which allows you to specify which columns (or expressions) you want to return under which column name. This approach allows you again to have all the logic (renaming) in one single place (namely the expression for the calculated table). In the parts about Power Query / M and SQL I will show you, how you can automatically rename all columns, without specifying each and every column, as we need to do in DAX.

```
[Order Date (DAX)] = /* calculated table */
SELECTCOLUMNS(
    'Date',
    "Order Date", [Date],
    "Order Year", [Year],
    "Order Month", [Month]
)
```

```
[Sales Date (DAX)] = /* calculated table */
SELECTCOLUMNS(
    'Date',
    "Sales Date", [Date],
    "Sales Year", [Year],
    "Sales Month", [Month]
)
```

Slowly Changing Dimensions

If you want to implement *Slowly Changing Dimensions*, you have to do this in a physically implemented data warehouse. In DAX you cannot update rows to keep track of changes and different versions. Usually, *Slowly Changing Dimension* means not extra effort in the world of DAX, as the rows in the fact table are already referencing the right version of the dimension table. Only, if your report user need to override the default version (= the version which was valid at the point in time the fact was collected), then you need to reach out to DAX and implement the logic via `CALCULATE` in your measures.

Figure 6-4 shows a report page with the following content:

- A slicer to choose the product.
- A slicer to choose the year, month, or day when the product had to be

valid. If a time range is selected (e. g. a year) then the version valid at the end of this period will be taken into account.

- Two Card visuals showing the last day of the period chosen (`Selected Product Version`) and the `Standard Cost` valid for this day.
- A Table visual showing columns
  - `Date`
  - `Product name`
  - `StartDate`, `EndDate` and `StandardCost` of the version of the product valid at the `Date`,
  - `Quantity` sold on that date
  - `Cost` as the result of the shown `StandardCost` times the shown `Quantity`
  - `Cost (Product Version)` calculated as `Product’s version Standard Cost` as shown at the top of the screen times `Quantity` sold on that day

Product

Multiple selections

Product Version

2023

2023-12-31

Selected Product Version

€ 39.2589

Product Version's Standard Cost

Date	Product	StartDate	EndDate	StandardCost	Quantity	Cost	Cost (Product Version)
2020-12-01	Sport-100 Helmet, Black	2020-01-01	2022-01-01	12.0278	27	324.7506	353.3301
2021-01-01	Sport-100 Helmet, Black	2020-01-01	2022-01-01	12.0278	56	673.5568	732.8328
2021-03-01	Sport-100 Helmet, Black	2020-01-01	2022-01-01	12.0278	86	1,034.3908	1,125.4218
2021-05-01	Sport-100 Helmet, Black	2020-01-01	2022-01-01	12.0278	162	1,948.5036	2,119.9806
2021-07-01	Sport-100 Helmet, Black	2020-01-01	2022-01-01	12.0278	10	120.2780	130.8630
2021-08-01	Sport-100 Helmet, Black	2020-01-01	2022-01-01	12.0278	65	781.8070	850.6095
2021-09-01	Sport-100 Helmet, Black	2020-01-01	2022-01-01	12.0278	28	336.7784	366.4164
2021-10-01	Sport-100 Helmet, Black	2020-01-01	2022-01-01	12.0278	92	1,106.5576	1,203.9396
2021-11-01	Sport-100 Helmet, Black	2020-01-01	2022-01-01	12.0278	74	890.0572	968.3862
2021-12-01	Men's Sports Shorts, L	2022-01-01	2023-01-01	24.7459	46	1,138.3114	
2021-12-01	Sport-100 Helmet, Black	2022-01-01	2023-01-01	13.8782	153	2,123.3646	2,002.2039
2022-01-01	Men's Sports Shorts, L	2022-01-01	2023-01-01	24.7459	41	1,014.5819	
2022-01-01	Sport-100 Helmet, Black	2022-01-01	2023-01-01	13.8782	238	3,303.0116	3,114.5394
2022-02-01	Men's Sports Shorts, L	2022-01-01	2023-01-01	24.7459	62	1,534.2458	
2022-02-01	Sport-100 Helmet, Black	2022-01-01	2023-01-01	13.8782	197	2,734.0054	2,578.0011
2022-03-01	Men's Sports Shorts, L	2022-01-01	2023-01-01	24.7459	41	1,014.5819	
2022-03-01	Sport-100 Helmet, Black	2022-01-01	2023-01-01	13.8782	134	1,859.6788	1,753.5642
Total					**4892**	**69,866.3581**	**58,194.7761**

Figure 6-4. A report page which gives the choice of the `Standard Cost` of which product's version should be used to calculate the `Cost (Product Version)`

For Product `Sport-100 Helmet, Black` a `StandardCost` of 12.0278 was valid in years 2020 and 2021 (`StartDate` 2020-01-01 and `EndDate` 2022-01-01). With beginning of 2022 the `StandardCost` rose to 13.7882. In the individual lines of the Table visual the `Cost` is calculated by multiplying `Quantity` with either of those two values (e. g. 27 * 12.0278 = 324.7506). In contrast, the value in column `Cost (Product Version)` is calculated as the individual `Quantity` by 13.7882 in all rows, as this is the standard cost valid for the selected version of the product (e. g. 27 * 13.7882 = 374.7114).

To implement this, we need the following parts:

1. A table containing the versions, where the report user selects from. As new versions can be created in any point in time, probably it is a good idea to use the date dimension (and possibly the time dimension) here.

   ```
   [Product Version] = 'Date' /* calculated table */
   ```

   Alternatively, you could also create a table containing all distinct `EndDate` values from the fact table. I decided against it here, as in a real-world scenario there could be a along list of those version, which will be possibly spread very unevenly over time, which makes scrolling down the

list a bit awkward. But it's totally up to you to exchange the reference to the `Date` table with `DISTINCT(Product[EndDate])`.
Resist to create a relationship from this table to e. g. the `StartDate` or `EndDate` of the `Product` table. Such a filter would not work as expected, as someone could select a date not existing as a `StartDate` or `EndDate`. Therefore, we will apply the filter over DAX in the measure where we are calculating `Cost (Product Version)`.

2. A measure for the selected product version:

```
[Selected Product Version] := MAX('Product Version'[
```

3. A measure to find the standard cost for the selected version, independent from the selected date. All the versions of the same product have the identical business key (`ProductAlternateKey`). Therefore, you need to remove any filter on the product table (as a filter e. g. on the name would be problematic, if the name changed over the versions) and add a filter on the `ProductAlternateKey` and find the product, for which the selected product version falls into the timespan of `StartDate` and `EndDate`. We need also take into account, that `StartDate` or `EndDate` could be empty, as the product's version is valid since ever or still valid.

```
[Standard Cost (Product Version)] :=
VAR ProductVersion = [Selected Product Version]
```

```
RETURN
SUMX(
    'Product',
    VAR AlternateKey = 'Product'[ProductAlternateKey
    VAR Result =
    CALCULATE(
        MIN('Product'[StandardCost]),
        ALL('Product'),
        'Product'[ProductAlternateKey] = AlternateKe
        ProductVersion >= 'Product'[StartDate] || IS
        ProductVersion <= 'Product'[EndDate] || ISBL
    )
    RETURN Result
)

[Cost (Product Version)] :=
SUMX(
    'Product',
    [Order Quantity] * [Standard Cost (Product Versi
)
```

Hierarchies

If you followed all best practices described in this book so far, then you already have denormalized all natural hierarchies in the dimension tables, as described in chapter “Denormalizing”. With the natural hierarchy

denormalized you have all levels of the hierarchy as columns in one single table. Adding them to a hierarchy is very easy.

Here we concentrate on parent-child hierarchies. They are very common, and we also need to store the names of all parents in dedicated columns. Read on if you want to learn how you can achieve this with DAX.

First, we create the materialized path. Luckily there is function DAX available:

```
'Employee (DAX)'[Path] = PATH('Employee (DAX)'[Employe
```

Then we need to dissect the `Path` and create a calculated column per (expected) level. Please add calculated columns for some extra levels in case the depth of the organigram (and therefore the path length of some of the employees) will increase in the future. To make creating these columns as convenient as possible, I put the level number (which should correspond with the name of the calculated column) into variable. Then you can just copy & paste this definition for each level and only change the name and the content of variable `LevelNumber`. `LevelNumber` is used as a parameter for `PATHITEM` to find the nth entry in the path. The found string represents the key of the employee and is stored in variable `LevelKey`. This key is then passed into `LOOKUPVALUE` to extract the full name of this employee and stored in variable `LevelName`. The latter is returned.

```
'Employee (DAX)'[Level 1] =
VAR LevelNumber = 1
VAR LevelKey = PATHITEM ( 'Employee (DAX)'[Path], Leve
VAR LevelName = LOOKUPVALUE ( 'Employee (DAX)'[FullNam
RETURN LevelName
```

You can already add all `Level` columns to a common hierarchy if you want. I created a Matrix visual, shown in Figure 6-5, with the hierarchy on the rows and measure `Sales Amount` in the value section. So far so good. As soon as you start expanding the upper levels you will see, that the `Sales Measure` is available for all (in my case seven) levels of the hierarchy, even when there is no employee related to the shown level. The result for the last available level is repeated for all sub-levels when they do not have their "own" value.

Hierarchies - Power BI Desktop

Search

Markus Ehrenmueller-Jensen

File Home Insert Modeling View Optimize Help External tools Format Data / Drill

Get data Sensitivity Publish

Level 1	Sales Amount
Ken J Sánchez	€ 80,450,596.9823
Brian S Welcker	€ 80,450,596.9823
Amy E Alberts	€ 15,535,946.2559
	€ 732,078.4446
	€ 732,078.4446
	€ 732,078.4446
	€ 732,078.4446
Jae B Pak	€ 8,503,338.6472
	€ 8,503,338.6472
	€ 8,503,338.6472
	€ 8,503,338.6472
Rachel B Valdez	€ 1,790,640.2311
	€ 1,790,640.2311
	€ 1,790,640.2311
	€ 1,790,640.2311
Ranjit R Varkey Chudukatil	€ 4,509,888.933
	€ 4,509,888.933
	€ 4,509,888.933
	€ 4,509,888.933
Total	€ 80,450,596.9823

Filters

Visualizations

Build visual

Rows

Employee Hierarchy

Level 1

Level 2

Level 3

Level 4

Level 5

Level 6

Data

nployee (DAX) Employee (PQ) Employee (SQL) Employee (DAX) Sales Employee (DAX) Sales without empty rows

Page 11 of 12 61% Update available (click to download)

Figure 6-5. The hierarchy expands to unnecessary levels with empty names and repeating `Sales Amount`.

In a good data model, you should take care of this problem. You need to add another column (to calculate on which level an employee is) and a measure to aggregate this column with `MAX`, another measure (to calculate on which level the measure is actually displayed) and tweak the existing measures to return blank in case an employee is shown in an unnecessary level (by returning blank in case the level the measure is displayed on is higher than the employee). The unnecessary level will not be displayed, if all measures only return blank.

We can calculate the level of an employee by counting the levels the path contains (by basically counting the separator character plus one). This gives us the position of an employee within the organigram. The lower the path length, the higher the position in the organigram, with the CEO having a path length of 1. Calculating this is much easier, than you might think, thanks to function `PATHLENGTH`. Calculating the maximum as a measure is then no real challenge, I guess:

```
'Employee (DAX)'[PathLength] = PATHLENGTH('Employee (D

[MaxPathLength] := MAX('Employee (DAX)'[PathLength])
```

We need also to count at which level the measure is. Here we need to check,

if the column, representing a certain level, is in the current scope of the visual or not. If it is, `INSCOPE` will return `TRUE`, which is implicitly converted to 1 in an arithmetic calculation. If it is not in scope, then `INSCOPE` will return `FALSE`, which is implicitly converted to 0 in an arithmetic calculation. In case you add columns for additional level, please remember to add them in the calculation of this measure as well.

```
[CurrentLevel (DAX)] :=
ISINSCOPE('Employee (DAX)'[Level 1]) +
ISINSCOPE('Employee (DAX)'[Level 2]) +
ISINSCOPE('Employee (DAX)'[Level 3]) +
ISINSCOPE('Employee (DAX)'[Level 4]) +
ISINSCOPE('Employee (DAX)'[Level 5]) +
ISINSCOPE('Employee (DAX)'[Level 6]) +
ISINSCOPE('Employee (DAX)'[Level 7])
```

Finally, we need to add a measure, in which we decide if a value has to be displayed or not.

```
Sales Amount (DAX) =
VAR Val = [Sales Amount]
VAR ShowVal = [CurrentLevel (DAX)] <= [MaxPathLengh (D
VAR Result =
    IF ( ShowVal, Val )
RETURN
    Result
```

TIP

Again, *Calculation Groups* can be of help here. You can create two *Calculation Group* items. One to just return the plain measure (" `SELECTEDMEASURE()` "), another where you copy and paste the code from measure `Sales Amount (DAX)` and replace " `[Sales Amount]` " with " `SELECTEDMEASURE()` ".

If you now exchange the `Sales Amount` measure with the newly created `Sales Amount (DAX)` measure, you get rid of the unnecessary empty levels of the hierarchy, as you can see in Figure 6-6.

Hierarchies - Power BI Desktop

Search

Markus Ehrenmueller-Jensen

File Home Insert Modeling View Optimize Help External tools Format Data / Drill

Get data Sensitivity Publish

Level 1	Sales Amount (DAX)
Ken J Sánchez	$80,450,597
Brian S Welcker	$80,450,597
Amy E Alberts	$15,535,946
Jae B Pak	$8,503,339
Rachel B Valdez	$1,790,640
Ranjit R Varkey Chudukatil	$4,509,889
Stephen Y Jiang	$63,320,315
David R Campbell	$3,729,945
Garrett R Vargas	$3,609,447
Jillian Carson	$10,065,804
José Edvaldo Saraiva	$5,926,418
Linda C Mitchell	$10,367,007
Michael G Blythe	$9,293,903
Pamela O Ansman-Wolfe	$3,325,103
Shu K Ito	$6,427,006
Tete A Mensa-Annan	$2,312,546
Tsvi Michael Reiter	$7,171,013
Syed E Abbas	$1,594,335
Total	$80,450,597

Filters

Visualizations

Build visual

Rows

Employee Hierarchy

Level 1

Level 2

Level 3

Level 4

Level 5

Level 6

Data

Employee (DAX) | Employee (PQ) | Employee (SQL) | Employee (DAX) Sales | Employee (DAX) Sales without empty rows

Page 12 of 12

61%

Update available (click to download)

Figure 6-6. The hierarchy expands to unnecessary levels with empty names and repeating `Sales Amount`.

Key Takeaways

Preferably, you push transformations as for as possible upstream, that means to Power Query, or, better, into the data source (e. g. a data warehouse). You can though do everything in DAX as well if you feel more comfortable. The only exceptions are semi- and non-additive calculations – for them there is no way around DAX. Take a look on what you learned in this chapter:

- Normalizing your fact tables involves steps to find candidates for dimensions, creating dimension tables as calculated tables via `SELECTCOLUMNS` and hide those colum ns in the fact table. Unfortunately, we cannot actually remove those columns from the fact table, because it would break the DAX code of the dimension tables.
- Denormalizing in DAX means to use `RELATED` to move columns over to the main dimension. Again, we cannot remove the referenced tables without breaking the DAX code. Therefore, we just hide these tables.
- I would recommend creating all calculations as DAX measures, as a starting point (and not as DAX calculated columns or as columns in the Power Query or in SQL). Carefully analyze the formula if it involves multiplication, because then you may need to use an iterator function to achieve the correct result.

- We can solve the problem of role-playing dimensions in two ways: Either adding (inactive) relationships to the data model and activating them via `USERELATIONSHIP` in the measures where we do not want to use the active relationship. Or we can add the dimension several times under different names and create standard relationships. Then no special treatment of your DAX code is necessary.
- Natural hierarchies are denormalized anyways in a star schema.
- Parent-child hierarchies need some extra love before we can use them conveniently in reports. You need to create some extra columns and measures.

About the Author

Markus Ehrenmueller-Jensen models data since over three decades. He teaches best practices at international conferences, webinars and workshops and implements them for clients in various industries. Since 2006 he is specialized in building Business Intelligence and Data Warehouse solutions with Microsoft's Data Platform (relational database engine on- and off-premises, Analysis Services Multidimensional and Tabular, Reporting Services, Integration Services, Azure Data Factory, Power BI dataflows, and – of course – Power BI Desktop and its predecessor PowerPivot for Excel). He is the founder of *Savory Data* (www.savorydata.com) and a professor for *Databases and Information Systems* at *HTL Leonding* (technical college, www.htl-leonding.ac.at) and holds several Microsoft certifications. Markus is the (co)-author of several books and was repeatedly awarded as a Microsoft Data Platform MVP since 2017. You can contact him via markus@savorydata.com. When Markus does not beat data into shape, he holds the beat behind the drum set in various musical formations.