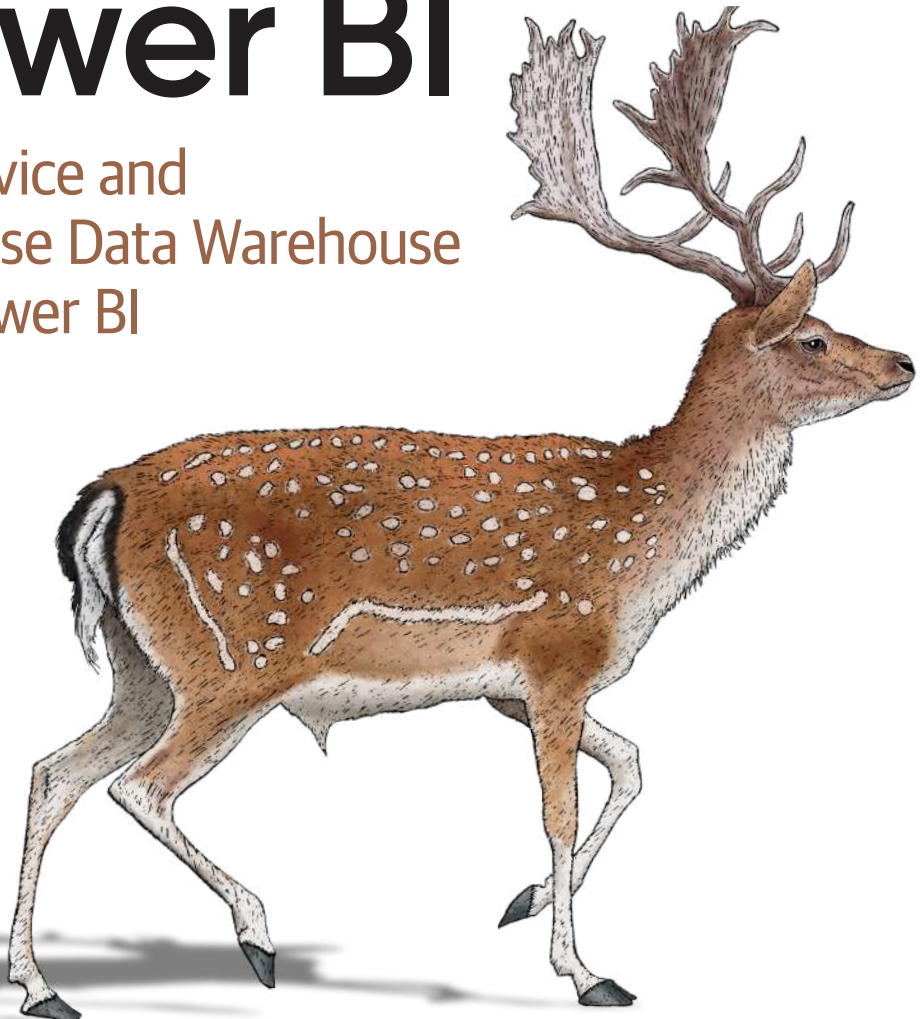


O'REILLY®

Data Modeling with Microsoft Power BI

Self-Service and
Enterprise Data Warehouse
with Power BI



Markus Ehrenmueller-Jensen

Data Modeling with Microsoft Power BI

Data modeling is the single most overlooked feature in Power BI Desktop, yet it's what sets Power BI apart from other tools on the market. This practical book serves as your fast-forward button for data modeling with Power BI, Analysis Services tabular, and SQL databases. It serves as a starting point for data modeling, as well as a handy refresher.

Author Markus Ehrenmueller-Jensen, founder of Savory Data, shows you the basic concepts of Power BI's semantic model with hands-on examples in DAX, Power Query, and T-SQL.

You'll learn how to:

- Normalize and denormalize
- Apply best practices for calculations, flags, and indicators, time and date, role-playing dimensions, and slowly changing dimensions
- Solve challenges such as binning, budget, localized models, composite models, and key value tables
- Discover and tackle performance issues via the data model
- Work with tables, relations, set operations, normal forms, dimensional modeling, and ETL

"This book is a comprehensive tutorial that covers the subject in language that is easy to understand yet thorough, concise, and accurate. Markus's mastery of the art and science of data modeling provides value for any data professional working with Power BI."

—Paul Turley
Microsoft Data Platform MVP

Markus Ehrenmueller-Jensen, founder of Savory Data, has worked as a project leader, trainer, and consultant for data engineering, business intelligence, and data science since 1994. He's a software engineer and professor at HTL Leonding (technical college), teaching databases and project engineering. He has several Microsoft certifications and is a Microsoft Data Platform MVP.

DATA

US \$69.99 CAN \$87.99

ISBN: 978-1-098-14855-3



[linkedin.com/company/oreilly-media](https://www.linkedin.com/company/oreilly-media)
[youtube.com/oreillymedia](https://www.youtube.com/oreillymedia)

Praise for *Data Modeling with Microsoft Power BI*

This excellent book will tell you why to “star schema all the things.” It explains in great depth why data modeling is important and provides ample examples. Reading this book will make your life as a Power BI developer easier.

—Koen Verbeeck, Senior BI Architect, Star Schema Aficionado,
Microsoft Data Platform MVP

Markus’s book *Data Modeling with Microsoft Power BI* provides a very good introduction to data modeling principles for an effective data model in Power BI as well as in Excel’s data model, Power Pivot.

The book is written in an explanatory way, using clear language that can be read by both novices and experts alike. It is very accessible and a must-have for those who want to learn more about data modeling.

I especially like Markus’s division of the book into a kind of matrix, where each of the five main sections is divided into four chapters dealing with the same four subtopics—understanding the data model, building a data model, examples from the real world, and performance optimization—which become more and more complex throughout the book so you gradually get more and more insight into the many facets of data modeling.

—Jørgen Koch, Innovate, Microsoft Power BI and Office
Enthusiast (SME), author and Microsoft Certified Trainer

Creating a fancy report and tinkering with DAX or M-Code in the times of AI is not hard—creating a high-performing model that will work is. Markus is a “Model Wizard” and has fixed more of my work than I would like to admit. This hands-on guide will give you a shot at mastering the model-building challenges ahead of you.

Although Power BI has evolved in leaps and bounds, the challenge of building a solid and performing model has not. Markus has seen endless environments and setups. I am a witness to his magic; he helped me fix broken models just by taking a quick look. With his new book he shares his knowledge with all of us and with his hands-on approach, he will guide you to becoming a Model Wizard yourself.

—Joel Ruh, *Digital Transformation Lead at SkyFrame*

Data Modeling with Microsoft Power BI is a must-read for anyone looking to master this powerful tool. Markus Ehrenmueller-Jensen has created an easy-to-read and fun guide that will help you unlock the full potential of Power BI.

—Carola Seyr, *Business Intelligence Team Lead at
Porsche Holding Salzburg*

Data Modeling with Microsoft Power BI

*Self-Service and Enterprise
Data Warehouses with Power BI*

Markus Ehrenmueller-Jensen

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Data Modeling with Microsoft Power BI

by Markus Ehrenmueller-Jensen

Copyright © 2024 Savory Data GmbH. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

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.

Acquisitions Editor: Michelle Smith

Development Editor: Shira Evans

Production Editor: Katherine Tozer

Copyeditor: Liz Wheeler

Proofreader: M & R Consultants Corporation

Indexer: WordCo Indexing Services, Inc.

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

June 2024: First Edition

Revision History for the First Edition

2024-06-11: First 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.

978-1-098-14855-3

[LSI]

*I want to dedicate this book to the best thing that ever happened to me: my
lovely children Clara (Alex) and Victor.*

Table of Contents

Foreword.....	xv
---------------	----

Preface.....	xvii
--------------	------

Part I. Data Modeling 101

1. What Is a Data Model?.....	3
Data Model	4
Basic Components	5
Entity	5
Tables	6
Relationships	6
Primary Keys	7
Surrogate Keys	8
Foreign Keys	9
Cardinality	10
Combining Tables	11
Set Operators	11
Joins	13
Join Path Problems	20
Entity Relationship Diagrams	25
Data Modeling Options	28
Types of Tables	28
A Single Table to Store It All	28
Normal Forms	29

Dimensional Modeling	33
Granularity	35
Extract, Transform, Load	36
Ralph Kimball and Bill Inmon	38
Data Vaults and Other Anti-Patterns	40
Key Takeaways	42
2. Building a Data Model.....	43
Normalizing	44
Denormalizing	45
Calculations	46
Flags and Indicators	47
Time and Date	47
Role-Playing Dimensions	48
Slowly Changing Dimensions	49
Type 0: Retain Original	49
Type 1: Overwrite	50
Type 2: Add New Row	51
Type 3: Add New Attributes	52
Type 4: Add Mini-Dimensions	53
Types 5, 6, and 7	53
Hierarchies	53
Key Takeaways	55
3. Real-World Examples.....	57
Binning	58
Adding a Column to a Fact Table	58
Creating a Lookup Table	58
Describing the Ranges of the Bins	59
Budget	60
Identifying the Granularity	60
Handling Fact Tables of Different Cardinality	61
Multi-Language Model	63
Key-Value Pair Tables	65
Combining Self-Service and Enterprise BI	67
Key Takeaways	67
4. Performance Tuning.....	69
Key Takeaways	71

Part II. Data Modeling in Power BI

5. Understanding a Power BI Data Model.....	75
Data Model	75
Basic Concepts	78
Tables and Columns	78
Relationships	88
Primary Keys	94
Surrogate Keys	94
Foreign Keys	94
Cardinality	95
Combining Tables	97
Set Operators	97
Joins	97
Join Path Problems	98
Entity Relationship Diagrams	100
Data Modeling Options	101
Types of Tables	101
A Single Table to Store It All	103
Normal Forms	106
Dimensional Modeling	107
Granularity	107
Extract, Transform, Load	108
Key Takeaways	108
6. Building a Data Model in Power BI.....	109
Normalizing and Denormalizing	109
Calculations	112
Time and Date	116
Turning off Auto Date/Time	116
Marking the Date Table	121
Role-Playing Dimensions	123
Slowly Changing Dimensions	127
Hierarchies	129
Key Takeaways	130
7. Real-World Examples Using Power BI.....	133
Binning	134
Lookup Table	134

Range Table	135
Budget	135
Multi-Language Model	139
Dimension Table for the Available Languages	139
Visual Elements	140
Text-Based Content	141
Numerical Content	142
Data Model's Metadata	143
UI of Power BI Desktop (Standalone)	146
UI of Power BI Desktop (Windows Store)	147
UI of the Power BI Service	148
UI of Power BI Report Server	148
Key-Value Pair Tables	149
Combining Self-Service and Enterprise BI	150
Key Takeaways	152
8. Performance Tuning in the Power BI Data Model.	153
Storage Mode	153
Partitioning	160
Pre-Aggregating	166
Composite Models	168
Dual Mode	169
Hybrid Tables	170
Key Takeaways	170
<hr/>	
Part III. Data Modeling for Power BI with the Help of DAX	
9. Understanding a Data Model from the DAX Point of View.	175
Data Model	175
Basic Components	176
Tables	176
Relationships	179
Primary Keys	180
Combining Queries	180
Set Operators	180
Joins	182
Extract, Transform, Load	185
Key Takeaways	186

10. Building a Data Model with DAX.....	187
Normalizing	187
Denormalizing	190
Calculations	190
Simple Aggregations for Additive Calculations	195
Semi-Additive Calculations	195
Re-create the Calculation as a DAX Measure	196
Time-Intelligence Calculations	198
Flags and Indicators	200
IF Function	200
SWITCH Function	201
SWITCH TRUE Function	201
Lookup Table	202
Treating BLANK values	202
Time and Date	203
Role-Playing Dimensions	206
Slowly Changing Dimensions	207
Hierarchies	210
Key Takeaways	214
11. Real-World Examples Using DAX.....	215
Binning	216
Lookup Table	216
Range Table	217
Budget	220
Multi-Language Model	222
Key-Value Pair Tables	229
Combining Self-Service and Enterprise BI	232
Key Takeaways	233
12. Performance Tuning with DAX.....	235
Storage Mode	235
Pre-Aggregating	235
Aggregation-Aware Measures	236
Key Takeaways	237

Part IV. Data Modeling for Power BI with the Help of Power Query

13. Understanding a Data Model from the Power Query Point of View.	241
Data Model	242
Basic Components	244
Tables or Queries	244
Relationships	248
Primary Keys	248
Surrogate Keys	249
Combining Queries	251
Set Operators	251
Joins	252
Query Dependencies	253
Types of Queries	255
Extract, Transform, Load	256
Key Takeaways	256
14. Building a Data Model with Power Query and M.	257
Normalizing	258
Column Quality	258
Column Distribution	259
Column Profile	260
Identifying the Columns to Normalize	261
Creating a Query per Dimension	265
Creating One Common Dimension Query	269
Denormalizing	270
Calculations	273
Flags and Indicators	275
Time and Date	279
Role-Playing Dimensions	284
Slowly Changing Dimensions	285
Hierarchies	286
Key Takeaways	295
15. Real-World Examples Using Power Query and M.	297
Binning	298
Create a Bin Table by Hand	298
Deriving the Bin Table from the Facts	299
Create a Bin Table in M	301

Create a Bin Range Table in M	307
Budget	308
Multi-Language Model	313
Key-Value Pair Tables	315
Using the GUI	316
Using M Code	319
Writing an M Function	320
Combining Self-Service and Enterprise BI	324
Key Takeaways	325
16. Performance Tuning the Data Model with Power Query.....	327
Storage Mode	327
Partitioning	328
Pre-Aggregating	329
Key Takeaways	331
<hr/>	
Part V. Data Modeling for Power BI with the Help of SQL	
17. Understanding a Relational Data Model.....	335
Data Model	335
Basic Components	336
Tables	336
Relationships	338
Primary Keys	338
Surrogate Keys	339
Foreign Keys	340
Combining Queries	341
Set Operators	341
Joins	344
Join Path Problems	351
Entity Relationship Diagrams	356
Extract, Transform, Load	357
Key Takeaways	359
18. Building a Data Model with SQL.....	361
Normalizing	362
Persisting into a Table	367
Creating a View	369
Creating a Function	370

Creating a Procedure	371
Creating a Filter Dimension	373
Denormalizing	375
Calculations	376
Flags and Indicators	379
Time and Date	383
Role-Playing Dimensions	385
Slowly Changing Dimensions	387
Type 0: Retain Original	388
Type 1: Overwrite	389
Type 2: Add New Row	392
Hierarchies	395
Key Takeaways	397
19. Real-World Examples Using SQL.....	399
Binning	399
Deriving the Lookup Table from the Facts	400
Generating a Lookup Table	401
Range Table	402
Budget	403
Multi-Language Model	404
Key-Value Pair Tables	408
Combining Self-Service and Enterprise BI	414
Key Takeaways	414
20. Performance Tuning the Data Model with SQL.....	417
Storage Modes	417
Table	418
Index	418
Compression	420
View	420
Function	421
Stored Procedure	421
Partitioning	421
Pre-Aggregating	429
Key Takeaways	430
Epilogue.....	431
Index.....	433

Foreword

No topic in the data industry is more debated than data modeling. It is the source of memes, T-shirts, and endless debates at conferences, and its demise has been predicted for years.

Yet here we are with a new book about data modeling. And it is sorely needed; data modeling is a foundational skill with many applications. It makes tough problems easier to solve, data easier to work with, and Data Analysis Expressions (DAX) easier to write. It improves performance and eventually saves costs. However, you must be willing to put in the work; it's necessary to start thinking about data modeling early in the process, whether you're designing a data warehouse, lakehouse, or semantic model in Power BI.

The data model is the cornerstone of your project. I have learned this from working with customers on all variations of analysis services over the years (from Power Pivot to SSAS and Power BI). With proper data modeling, you won't have to resort to as many DAX gymnastics. A good data model simplifies your calculations.

I've known Markus for many years and always enjoy his sessions at conferences. He explains tough topics in a simple manner, and this book is no exception.

In *Data Modeling with Microsoft Power BI*, Markus explores the many facets and long history of data modeling (who doesn't have the Kimball data warehousing book on the shelf?): how we need to think about data, and how we can translate requirements into entities and attributes. Markus does a great job applying these theoretical practices to real life.

How do these data modeling practices help you with everyday Power BI and SQL challenges? Markus explains the basics of data modeling in Power BI by looking at tables, relationships, and analysis of data granularity. He then shows how to translate common requirements like role-playing dimensions, slowly changing dimensions, binning, and translations into SQL, DAX, M, or in the model itself so you can use them at any step of your project.

The lessons in this book are very valuable, helping you simplify your day-to-day work as a data engineer, and it all starts with the model.

— *Kasper de Jonge*
Principal Program Manager at Microsoft Fabric

Preface

Welcome to this journey into data modeling concepts and practical examples for Power BI, including DAX, Power Query and T-SQL. This book is your companion on your journey to gain a comprehensive understanding about the steps needed to make building reports in Power BI Desktop and Power BI Report Builder, and creating measures in DAX, easier.

Power BI supports a **wide variety of data sources** (covering databases from different vendors, like Microsoft, Oracle or Teradata; flat files, like CSV, text, or Excel; web services like an https link to a web page, etc.). The only way to get data into Power BI is through Power Query. It's best practice to add calculations as (*explicit*) *measures* in DAX (as opposed to *calculated columns* in DAX or as columns in Power Query or in the data source). Creating *calculated tables* in DAX should be an exception; depending on your skills and preferences, you will implement transformations to shape the data model either in Power Query (in the user interface or by writing code in the M language) or in the data source. For example, in the case of a relational data warehouse implemented in Microsoft's relational database engines, you might use T-SQL in the data warehouse, as laid out in **Figure P-1**.

The first part of this book, which is written in an agnostic way, introduces the necessary concepts in a general way: you can apply this to any analytical system. The second part of the book explains the properties of a data model in Power BI. The rest of this book addresses DAX, Power Query, and SQL.

The book is designed for you, the reader, to have an individual experience based on your knowledge. You may not know DAX, Power Query, *and* SQL, but you may have familiarity with one or two of them; you can pick and choose to fill in gaps in your knowledge. Maybe you need a refresher on the composition of a data model. **Part I** has you covered. Maybe you struggle with dealing with a bunch of Excel files from which you need to create reports? The part on Power Query will be your starting point. Maybe your task is to build a data warehouse to which other people connect

with Power BI Desktop? Then the part about SQL will present you with solutions to typical problems.

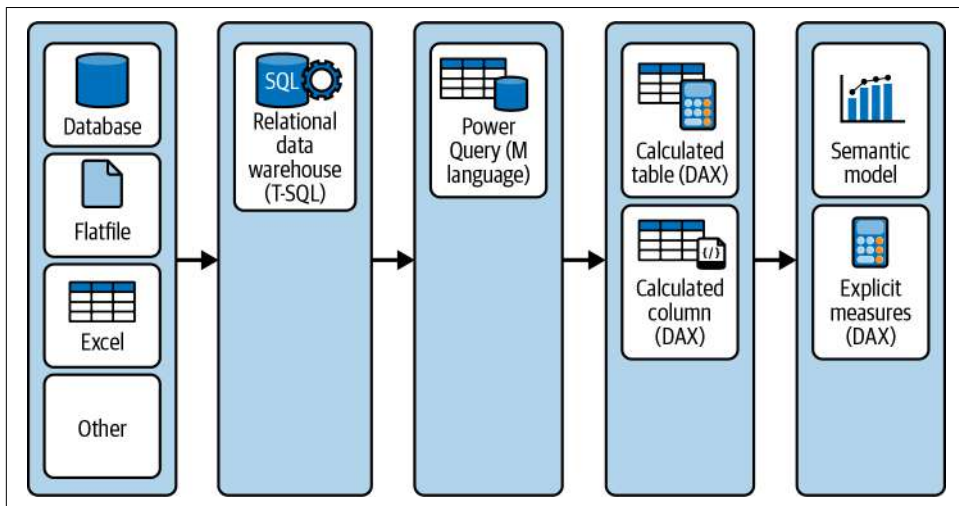


Figure P-1. Power BI data-shaping architecture

Data modeling is definitely the single most underestimated task when working with Power BI Desktop. It is a crucial part in your steps, from raw data to business intelligence and analytics. Decisions made during data modeling will influence how much detail your reports can show, how user-friendly the database or semantic model is for creating reports and analysis, and how easy it is to add more data and implement calculations on existing data. Wrong decisions at the start are very expensive to fix later, as changes to the data model will break existing reports. I speak from experience: I had to learn this the hard way—and I see also other people struggling with the repercussions almost every day in my work as a trainer and consultant. This book is your guide to getting data modeling right from the beginning.

You will learn that it is less important how complex the steps done in the “back end” (DAX, Power Query, or T-SQL) are, as long as the result is an easier-to-understand and easier-to-use data model for the user who creates reports and does analytics based on this data model: report authors, business analysts, data scientists, etc. These steps can be as simple as changing technical names (e.g., CSTNM4711) into user-friendly names (e.g., Customer Name) or as complicated as combining or splitting tables into a whole new structure. You will learn how to add calculations to the data model and enrich plain data with metadata (hierarchies, translations, etc.). This book is full of practical examples from challenges I faced over more than 25 years in the field. Keep in mind that the essential goal is to remove the burden from the report creator.

A common analogy for data transformation is a restaurant. As the restaurant's customer, you expect the dishes served attractively arranged on plates so you can enjoy the meal with only common tools (spoon, knife, fork, chopsticks, or maybe your fingers). To make this happen, the restaurant not only reserves significant space and resources in the kitchen but also owns expensive tools (convection oven, broiler, sous-vide cooker, blender, etc.) and employs formally trained and skilled people (cooks) to control those devices to transform the raw ingredients into the dishes. Think of the people using a data model to create reports and do analytics as restaurant guests: they will prefer to have all information presented in an easy-to-digest (pun intended) form, which they can consume with common tools (Power BI, Excel, etc.). Set yourself into the role of a cook who puts all her experience together to create a data model that invokes the appetite to consume it.

Are you a data cook? Read on!

Who Is This Book For?

Are you accessing data in Power BI Desktop and interacting with it via visuals? Did you gather the most important data into one Power BI Desktop file so others can build reports on it? Are you in charge of a data warehouse and want to make sure that the data model is optimized for usage in Power BI Desktop? If you answer “Yes” to any of these questions, then this book is for you.

The primary audience is the enthusiastic Power BI report creator who wants to apply best practices in the data model for performant reports and easy DAX calculations. The secondary audience is the IT Pro who wants to support report creators with a connect-and-go data source (a data model created in Power BI Desktop). You should be comfortable with creating reports in Power BI Desktop and have a basic understanding of at least DAX, Power Query/M, or SQL, so you can follow the code examples provided in this book.

Throughout the book, you'll not only learn about the data modeling options in Power BI but also about modeling options in other tools, so you can create a data model that is optimal for Power BI. This is covered in **Part V**.

Why is it worth it to read (and write!) a whole book about data modeling? The next section delivers the answer.

What Is Data Modeling?

The challenges of storing data in different logical formats are as old as data itself. Before electronic computers were invented, data was put into different physical files and structured in physical folders, filling the shelves of big cabinets or even whole rooms or basements. Optional indexes (alphabetically ordered small cards with important terms and a reference on which shelf in which folder and file the information can be found) allowed you to scan the data not only by its physical order but by different tags. This terminology remains, but data is now stored in files and folders on hard disks, with additional indexes speeding up reading access.

Different approaches were invented and discussed over history to make for easy storage (e.g., avoiding redundancy and standardization of physical data storage) and easy read-access (e.g., indexing and reintroducing of some redundancy to speed up access to data). The concept of relational databases (e.g., SQL Server and Azure SQL Database) goes back to the year 1970. Dimensional modeling is even older—but nevertheless still very useful today.

To master Power BI, you need to master data modeling, because Power BI is a model-oriented analytics tool (as opposed to some other tools on the market). The next section gives you an overview about what parts Power BI is composed of and where you define the data model.

What Is Power BI?

Power BI is not a single tool but a whole suite of tools that became widely available in 2015. In case you are new to Power BI, I will give you a brief overview of the different tools:

Power BI Desktop

Power BI Desktop is the full client with which you can achieve a lot of tasks (see **Figure P-2**). You connect to data sources, clean and transform data (with Power Query), develop a data model (in the *Model view*), and create reports (in the *Report view*). This is a tool you will spend a lot of time with when re-creating the examples in this book. All examples and screenshots in this book covering Power BI data modeling, DAX, and Power Query are based on Power BI Desktop. Files created with Power BI Desktop have the extension `.pbix` or `.pbip`. Power BI Desktop is free of charge—you need no sort of license, and you can skip the sign-in step when the tool prompts you.

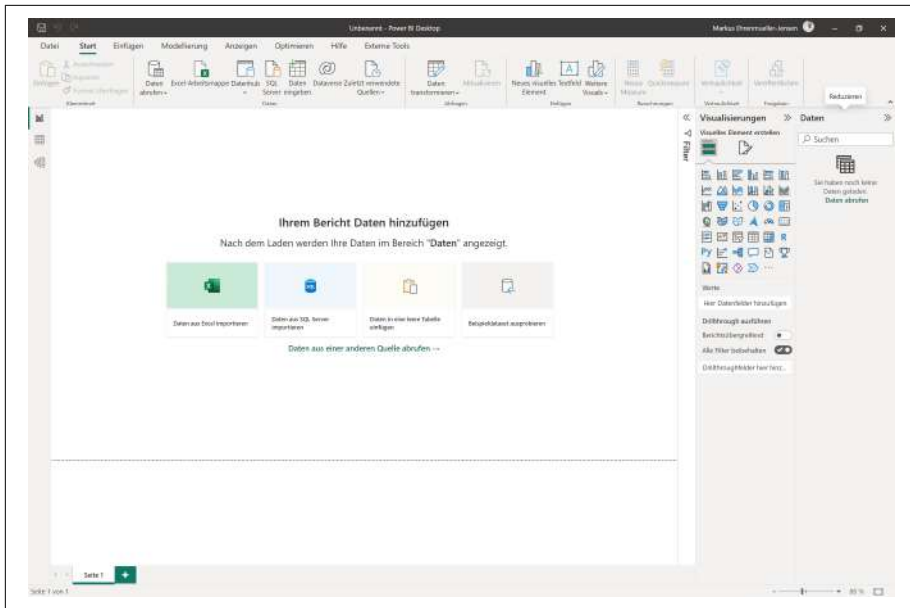


Figure P-2. The start screen in Power BI Desktop

The Power BI service

The Power BI service is where you host files you have created with Power BI Desktop so that others can consume a report or create a new report based on your data model. The Power BI service offers additional features (like Metric, Dashboard, Power BI App, Analyze in Excel, Export to Excel, etc.) not available in Power BI Desktop. The Power BI service is hosted in Microsoft's cloud data centers and is available with an internet browser of your choice via the **Power BI service** (see Figure P-3). You can also edit reports directly in the Power BI service, and Microsoft is working hard to enable editing of data models available in the Power BI service. At the time of writing, this lacks important features (like version control and collaborative editing)—so I'd currently recommend relying on Power BI Desktop instead. The Power BI service comes with **different licenses**.

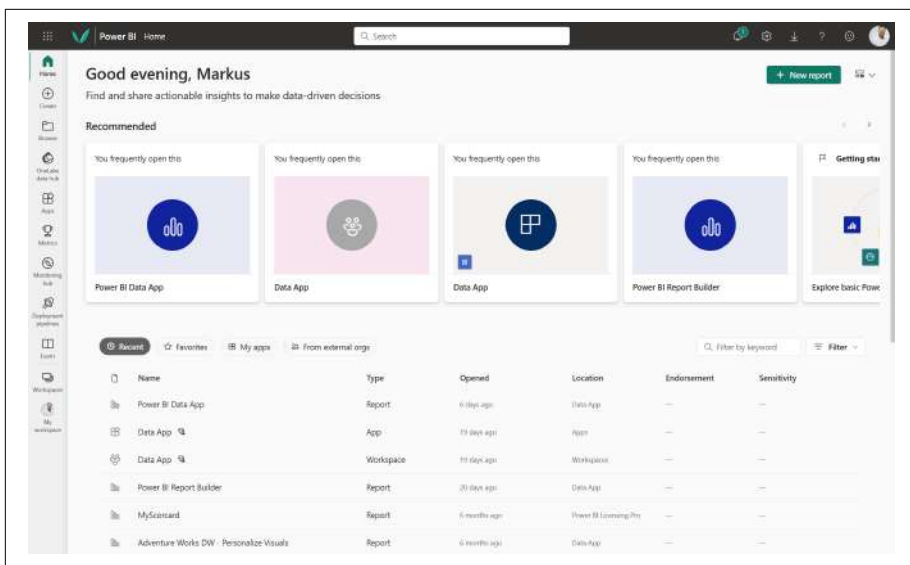


Figure P-3. The Power BI service says hello

Power BI Report Server

Power BI Report Server (see [Figure P-4](#)) is **an alternative to the Power BI service**, which you can install on your own premises. Power BI Report Server comes with a limited feature set, and new versions are released (only) three times a year. You need to use a matching version of Power BI Desktop (called *Power BI Desktop for Report Server*, which shows the month and year in the title of the application) when you intend to publish a Power BI Desktop report on a Power BI Report Server, as the monthly released version of Power BI Desktop might contain artifacts not compatible (yet) with the Power BI Server you are using.

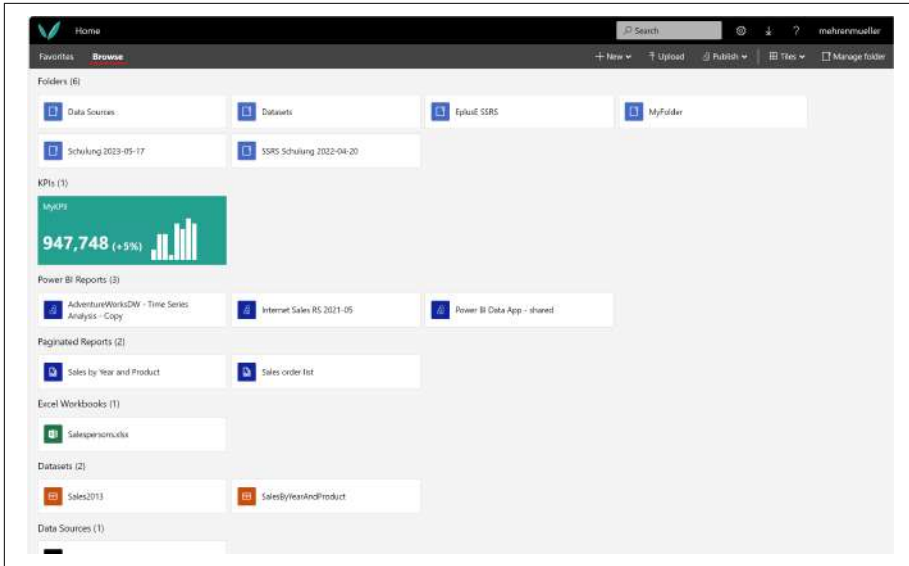


Figure P-4. The Power BI Server says hello

Power BI Report Builder

When you need to create pixel-perfect reports, with lists of data covering several pages, or you need to export in file formats not available for reports created with Power BI Desktop, then Power BI Report Builder is the tool for you (see Figure P-5). Power BI Report Builder is free of charge.



If you intend to publish paginated reports on a Power BI Report Server (as opposed to the Power BI service) you need to use *SQL Server Report Builder* instead of *Power BI Report Builder*.

You cannot create data models in (any version of) Report Builder.

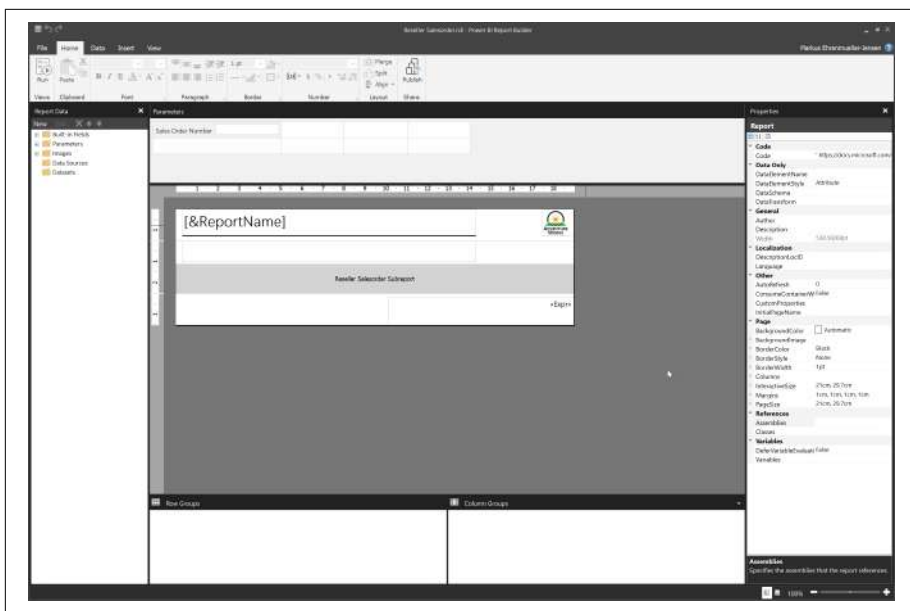


Figure P-5. Create paginated reports with Power BI Report Builder

The Analysis Services tabular model

You can think of Analysis Services tabular (which is available as Azure Analysis Services and as SQL Server Analysis Services tabular) as Power BI Desktop stripped from the report creation feature and reduced to the table view, the Model view, and Power Query. It shares the same storage engine (*VertiPaq*) and modeling capabilities as Power BI Desktop, and you can connect any reporting tool (including Power BI Desktop and Power BI Report Builder) to Analysis Services to create reports and analytics. You develop and deploy such a database with Visual Studio (see [Figure P-6](#)). Some of my customers use Azure Analysis Services instead of the Power BI service to host data because they can scale the costs for data storage at a more granular level (compared to the Power BI licensing costs); others use SQL Server Analysis Services because they cannot or do not want to host their data in the cloud.

Microsoft's vision is to make Power BI a super-set of Analysis Services tabular; with the announcement of **Microsoft Fabric** at the Build conference in May 2023, Microsoft made a big step toward it. Fabric will also allow scaling costs at a more granular level, compared to Power BI's licensing.

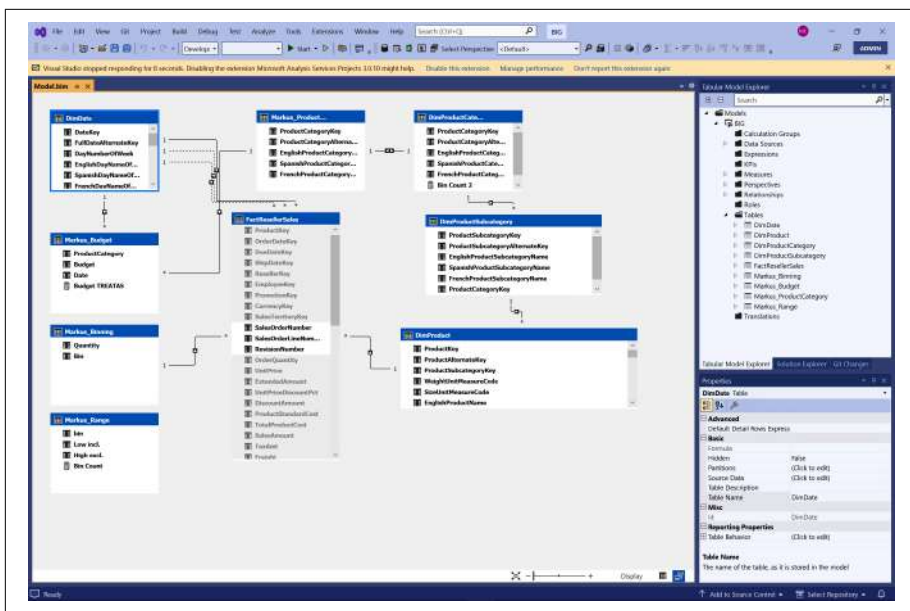


Figure P-6. The Analysis Services tabular database definition in Visual Studio

Let's talk about why you should care so much about the data model you build in Power BI Desktop.

What Is So Special About a Power BI Data Model?

Power BI is very versatile when it comes to the shapes of data models you can use (how you spread information between the tables or if you combine everything into a single table). But don't step into a trap here: the storage engine behind Power BI (called VertiPaq) and the language to define formulas for measure (DAX) are optimized for a certain shape called a *star schema*. This book is your guide to understand what a star schema is, why it is so important to take the time to transform the table(s) of your data source(s) into this shape and, most importantly, how to actually implement these steps.

In a nutshell, *star schema* is a term used for a data model where you have a fact table in the middle (forming the center of the star) surrounded by dimension tables (Figure P-7). I know, you need a lot of imagination to see such a star—but still, this concept is important and useful when building analytical systems in general, and especially when it comes to Power BI.

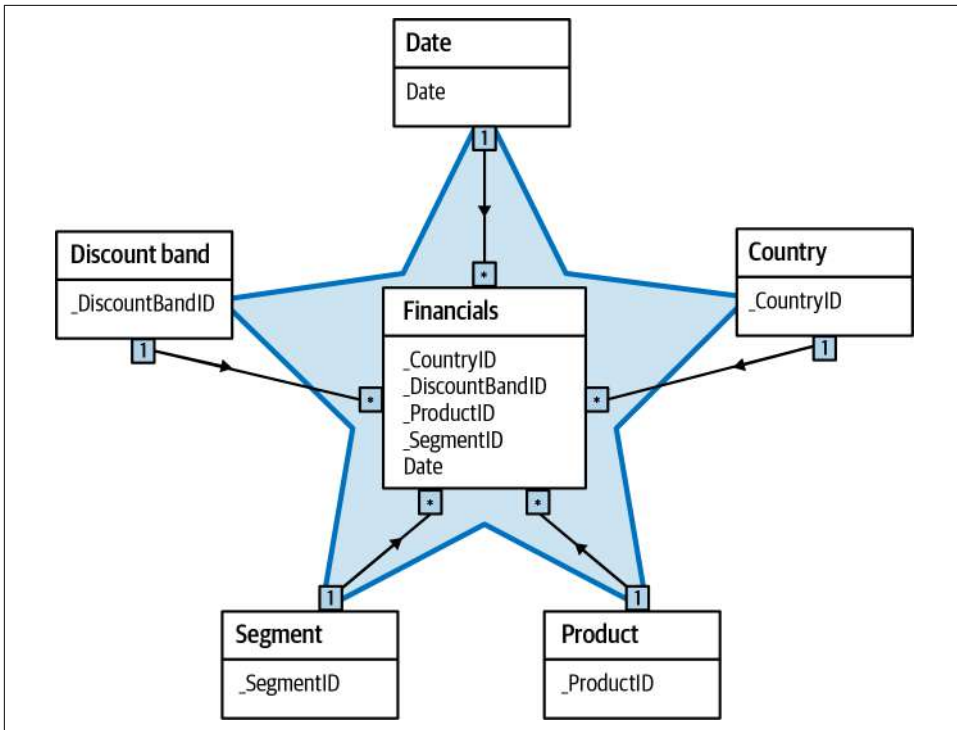


Figure P-7. A star schema

In a perfect world, the tables in the data source would already be in the shape of a star schema. If this is true for all your data sources, I consider you really lucky—and you can stop reading this book and ask for a refund. The mere mortals who are not as lucky have several options, which I discuss in this book. You can build a data warehouse layer (e.g., in form of a relational database or a data lakehouse). The SQL language will be your tool of choice. Or you can use Power BI's Power Query to reshape the tables. The third option is DAX, which is discussed in the next section.

What Is DAX?

Data Analysis Expressions (DAX) is a formula expression language used to create calculated columns, measures, calculated tables, and row-level security and to write queries. As you will learn, it's important to master DAX, as certain types of calculations can only be done in DAX (and not Power BI's data source or Power Query).

To reshape a data model into a star schema, you must move information from one table into another via calculated columns or create new tables as calculated tables. You'll also learn how to use DAX when you hit limits of Power BI's data modeling capabilities.

Overall, DAX isn't my first choice as a data-shaping tool if I can solve a problem with Power Query. Whenever possible, I push transformations into Power Query or, if available, into the data warehouse layer because it's best practice to push transformations as far "upstream" in the data processing pipeline as possible, to increase reusability of a transformation. A transformation in Power Query can be pushed into a Power BI dataflow to re-use it in different data models. Transformations in DAX are tightly applied to the data model they reside in. Another reason is that in Power Query and SQL, I can shape the data *before* I actually load it into Power BI, while in DAX I can only add calculated columns and tables on top of a model; any suboptimal part still occupies resources in my data model.

If you don't feel ready for Power Query (or the M language) or if you don't have a data warehouse at hand, then modeling the data in DAX is way better than not modeling your data at all. And remember, some problems can only be solved with a DAX measure.

What Is Power Query?

As the name suggests, Power Query is a tool to create queries you send against your data source(s). At the time of writing, Power Query has connectors to over 120 data sources, from flat files like CSV, XLS, or JSON to relational (SQL Server, Oracle, DB2, Teradata, etc.) and analytical databases (Analysis Services). Via a graphical user interface (GUI), you can clean and transform the original tables: renaming tables and columns to give them a more user-friendly name, removing unnecessary columns, and adding new tables and columns based on the existing information to shape a better data model.

All steps you apply in the GUI are added as lines of code to a Power Query script (called M for short), similar to the macro-recorder in Excel. Through the course of this book, you will learn how to use the GUI and when to edit the resulting query. Any time you refresh the content of a table, this script is executed and, therefore, all transformation steps are applied to the new data as well. That's why I ask people to hand me over "raw" CSV or Excel files (in cases when files are the preferred data source). There's no need to put effort into shaping the content manually every time they send me the new data if I can implement transformations once and re-apply them during the refresh of my data model.

Power Query (and therefore scripts in its mashup language, M) is not only available in Power BI Desktop but as Power BI dataflows and mashup tasks in Azure Data Factory as well. But maybe you want to push transformation further up the data stream (and if you ask me, you should). That's why there's so much content in this book about Azure SQL DB and T-SQL.

What Is SQL?

SQL is an acronym for *Structured Query Language*. In the context of Microsoft, “SQL” can refer to Microsoft’s relational database engines offerings as well. Azure SQL DB is Microsoft’s cloud offering for relational databases, which is available for on-premises’ usage under the name Microsoft SQL Server Database Engine.

In this book, Azure SQL DB is used as an example of how to model your data outside of Power BI. I still consider this best practice: model your data in a database, and model your data as early as possible, or in other words, in a data warehouse layer. It is less important how you implement this layer: as a physically hosted relational database, as a data lakehouse, or in any other data store, like somewhere in Microsoft’s Fabric.

Pushing the transformations as far as possible toward the data source makes it easier for people down the data stream to re-use it. If you wait until the last moment (that is, to determine the tool you use to show your data, e.g., Power BI Desktop, Power BI Report Builder, Excel or any other tool your end users might use to access the data and do their analytics) then you have accumulated a technical debt. The end users are then responsible for cleaning the data and bringing it into a useful shape. Due to lack of education, they might fail—leading to overcomplicated reports and possibly wrong numbers (you will learn about this problem in [Chapter 5](#)). On top of that, all the work is then redone or copied over into the next file—adding the problem of duplicated versions of these overcomplicated reports. Soon, you could end up in so-called Excel Hell, where nobody has a complete picture of the different versions of logic applied to the data. On the other hand, a data warehouse layer guarantees a single version of the truth.

Part V concentrates on solutions built with SELECT statements and SQL’s procedural extension, T-SQL. I use the SQL dialect available in Azure SQL DB and SQL Server Database Engine. There is some chance that simple SELECT statements in this book will also run on other relational databases, or even *NO-SQL* databases. There is almost no chance that the procedural extensions (loops, functions, procedures, etc.) will work without any change on other database management systems. You’ll have to find a way to migrate the code to your destination system if you aren’t using Microsoft’s SQL-based databases.

T-SQL as a language is quite stable in terms of how rarely new extensions are made or what parts are deprecated. Power BI is a different beast—new versions are released every month.

A New Release Every Few Weeks

The team behind Power BI and its tools and services at Microsoft is very busy delivering new versions of Power BI Desktop every month and rolling out changes in the cloud-based services weekly. This is a challenge for everybody: the UI changes, icons are redesigned, and buttons are moved to different places. This is also a challenge for every book project: some of the screenshots may be outdated when you read this book. Therefore, I include only portions of the screen, when sufficient. In many places, I also link to [Microsoft's official documentation](#), which Microsoft and the community keeps up-to-date; you can double-check it if your Power BI Desktop looks different.

The general concepts on how to create an optimal data model for Power BI haven't changed much in the past, and therefore, there is hope that this knowledge is here to stay and will help you in the future as well. Read on to learn how I divide all the necessary knowledge and skills into digestible portions.

How to Read This Book

The book is organized into five parts—five books for the price of one:

Data Modeling 101 (Part I)

The first part gently introduces all the theory and concepts and teaches you why data modeling matters. It covers all the content in a practical but tool-agnostic way. Look at this part as a “reader’s digest” version of Ralph Kimball and Margy Ross’s *The Data Warehouse Toolkit*, Third Edition (Wiley, 2013) and Christopher Adamson’s *Star Schema: The Complete Reference* (McGraw Hill, 2010).

Data Modeling in Power BI (Part II)

The second part covers data modeling features of Power BI Desktop, where I show you how to apply all the theory (from the first part) in concrete examples. I guide you around the menu and ribbon and into the properties of a data model.

Data Modeling for Power BI with the Help of DAX (Part III)

This part shows how you can bring information from data source(s) into the necessary shape (discussed in [Part II](#)) with the help of DAX.

The advantage of using DAX is that you need to learn it anyway when you want to master Power BI (there’s no way around writing calculated measures in DAX for anything but very simple data models). Another is that changes in the formula will be calculated immediately after you press OK, without accessing the data source. The disadvantage of using DAX to reshape a data model is that the result of calculated columns and calculated tables will occupy disk space when you save the file and memory when you open it. Another disadvantage is that

your data model will contain both the original shape and the reshaped version of the data model (therefore occupying an unnecessary amount of space).

DAX's main purpose is to define measures and not to shape data, but some of the examples still provide additional insight about what you can achieve with DAX. Learning the full capabilities of the DAX language will also help you use complex DAX measures when you for some reason are not able to solve the problem in the model itself.

Data Modeling for Power BI with the Help of Power Query (Part IV)

This part shows how to bring information from data source(s) into the necessary shape with the help of Power Query's UI and the M language. You'll see that you can come a long way with the UI alone before handling code in M.

Unfortunately, M is very different from DAX. For one thing, in M you apply transformations based on the the result of a previous step; in DAX you add content on top of existing content of the data model. M is also case sensitive for keywords and the content of columns, and DAX is case insensitive. M is similar to F#, whereas DAX is similar to Excel formulas. The two can hardly be compared to each other. As you will see in this part, M as a language is much better suited to the task of data shaping than DAX.

When the data I'm loading isn't already in the desired shape, Power Query and M are my tools of choice. Only the final result of transformations made in Power Query are loaded into the Power BI data model, avoiding unnecessary tables and columns. While working with Power Query, you need active access to the data source.

Data Modeling for Power BI with the Help of SQL (Part V)

This part shows how you can bring the information from data source(s) into the necessary shape with the help of SQL and T-SQL. Despite trends to gather data in lakes, I still strongly believe that an enterprise organization needs a central place where someone takes care of the data; a place where natural (and dirty) data is cleaned and brought into the right shape. Whether this place is a physical database or "just" views on some data store is not so important. But it is important that you have such a (data warehouse) layer instead of putting the burden onto the end user and the tools they use.

If you're eager to learn about relational databases or to make the life of Power BI users in your organization easier, then this part of the book is especially for you. In Microsoft Fabric you can access your data residing in a data lake.

Each part has four chapters, which cover the following topics:

Understanding a data model

These chapters are about the basic terms and concepts. If you're new to data modeling, ensure you read and understand these chapters. If you already have some background in data modeling, you might quickly check these chapters to refresh your memory. Look out for the key takeaways at the end of each chapter.

Building a data model

These are all about the meat-and-potatoes of data modeling. These chapters discuss problems and solutions to common problems. I'm convinced that you will face at least some of those on your journey of refining your data into usable information.

Real-world examples

For these chapters, I address advanced challenges. For simple data models, the problems discussed here might not be an issue. You might enjoy a break on your first reading here and return when you find yourself facing one of the issues. These chapters dive deeper into data modeling, DAX, Power Query, and SQL and cover not-so-common features you might need to solve a certain problem. These chapters are aimed for the advanced data cook.

Performance tuning

Every part closes with a chapter on performance tuning, which is most often the last step in the whole journey of data modeling. If you start by learning to follow all best practices pointed out in this book, you will have plenty to do before report performance will be an issue. For really large implementations (I'm talking about billions of rows of data), though, even following these best practices won't be enough. Then it's time to dive into the technical layer of data modeling and learn how to tune the available storage modes (Import, DirectQuery, live connection, and Direct Lake) to your advantage.



The whole book, and therefore also the chapters about performance tuning, concentrates on data modeling and not so much on how to write performant code in DAX, Power Query or SQL.

All the parts and topics are brought together in 20 chapters (see [Figure P-8](#)). They progress in complexity and difficulty, allowing you to read the book cover-to-cover. Or you could jump over DAX or ignore SQL, for example, if you intend to use Power Query to solve your challenges. Imagine you jump to [Chapter 10](#) and discover that you're not sure why you should build a date table for your Power BI data model. You could jump to [Chapter 6](#) to find out. You might also review the general concept of a date table (and why it's useful in general and for analytical systems built with other tools) in [Chapter 2](#). Or maybe you decide against building the date table in DAX and

explore other solutions. You could find the same implementations done in Power Query in [Chapter 14](#) and in SQL in [Chapter 18](#). [Figure P-9](#) illustrates this journey.

Chapters	Parts				
	Part 1 Data modeling in general	Part 2 Power BI	Part 3 DAX	Part 4 Power Query	Part 5 SQL
Understanding a data model	Chapter 1	Chapter 5	Chapter 9	Chapter 13	Chapter 17
Building a data model	Chapter 2	Chapter 6	Chapter 10	Chapter 14	Chapter 18
Real-world examples	Chapter 3	Chapter 7	Chapter 11	Chapter 15	Chapter 19
Performance tuning	Chapter 4	Chapter 8	Chapter 12	Chapter 16	Chapter 20

Figure P-8. Chapter overview

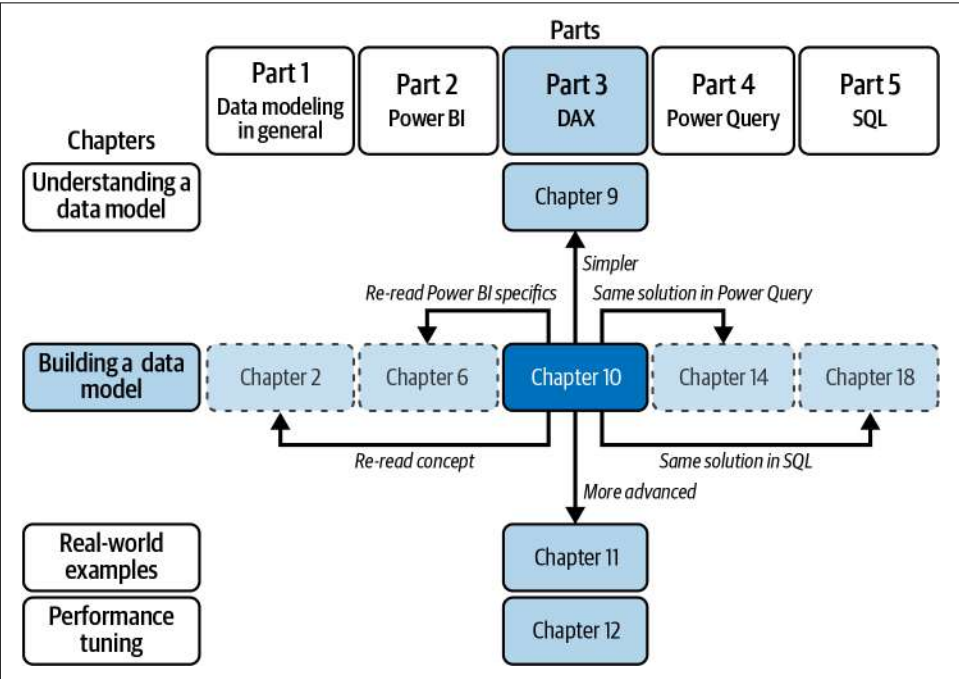


Figure P-9. Example navigation

Ready to begin? Ensure that you have the necessary software installed.

Installing Necessary Software

You need to install the following software to open the demo files I’ve provided and replicate the exercises described in this book:

Power BI Desktop

To open the *.pbix* files, you need to either install the Windows Store version from the **Power BI Desktop Store** or download the installation files from **Power BI Desktop installer**. Make sure to download and install the *.msi* file regularly, or install Power BI Desktop from the Microsoft Store, which will update automatically.

SQL Client

For the exercises in SQL, I use SQL Server Management Studio (SSMS), which you can download and install from **Download SQL Server Management Studio (SSMS)**. Alternatively, you can use Azure Data Studio, which you get from **Download and install Azure Data Studio**, or you use your preferred SQL client.

SQL Server or Azure SQL DB

For the exercises in SQL, you need access to one of the relational databases available from Microsoft: either SQL Server (installed on your premises) or Azure SQL DB (software-as-a-service in Microsoft’s cloud platform). You can **download SQL Server from Microsoft**. Use either the Express or Developer edition. Both are free of charge and sufficient for the exercises.

Alternatively, you can **sign up for an Azure SQL DB**. For the exercises, a free trial access will be sufficient.

Write access to database “AdventureWorksDW”

Most of the examples are based on a data warehouse schema of the fictitious company I call “Adventure Works,” a sport retailer that earns the majority of its revenue through selling bikes on three continents either via resellers or directly over its web shop. To create the necessary database objects (tables, views, procedures, functions, schemas), you need write access to the database “AdventureWorksDW” (the one with the “DW” in the suffix of the name, not: “AdventureWorks” or “AdventureWorksLT” or “Adventure Works OLTP”). You can find the backup file and a description of how to install it on SQL Server here: **AdventureWorks sample databases**.

To install database “AdventureWorksDW” as an Azure SQL DB, you can use the **“Data-tier Application” (also known as a BACPAC file)**. Install this BACPAC file via SQL Server Management Studio (SSMS). Right-click Databases and choose “Import Data-tier Application,” as shown in **Figure P-10**.

Provide the folder and file names of where you downloaded it (**Figure P-11**).

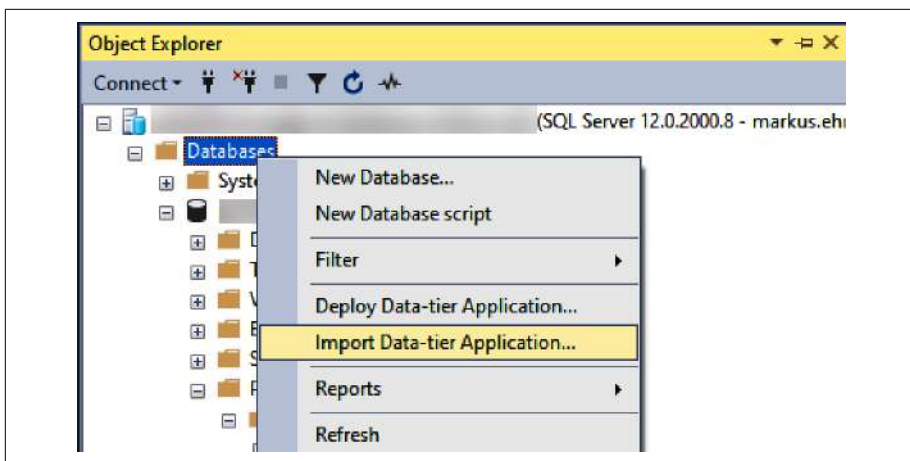


Figure P-10. Import Data-tier Application in SQL Server Management Studio

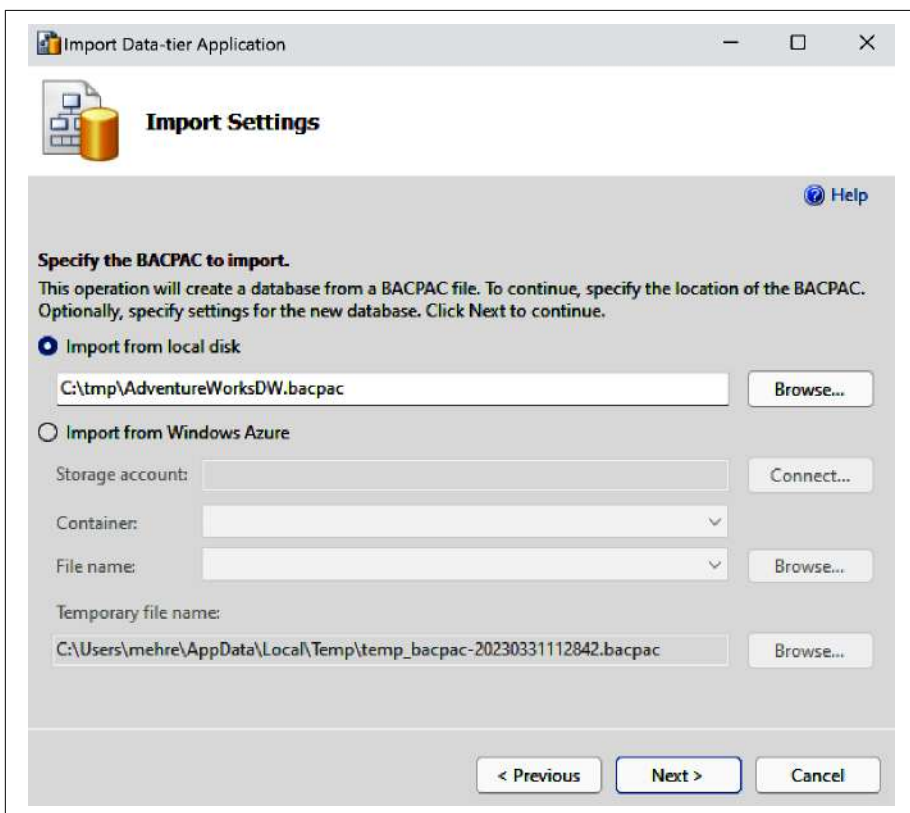


Figure P-11. Provide the name of the folder and file

The smallest/cheapest edition will be sufficient (Figure P-12).

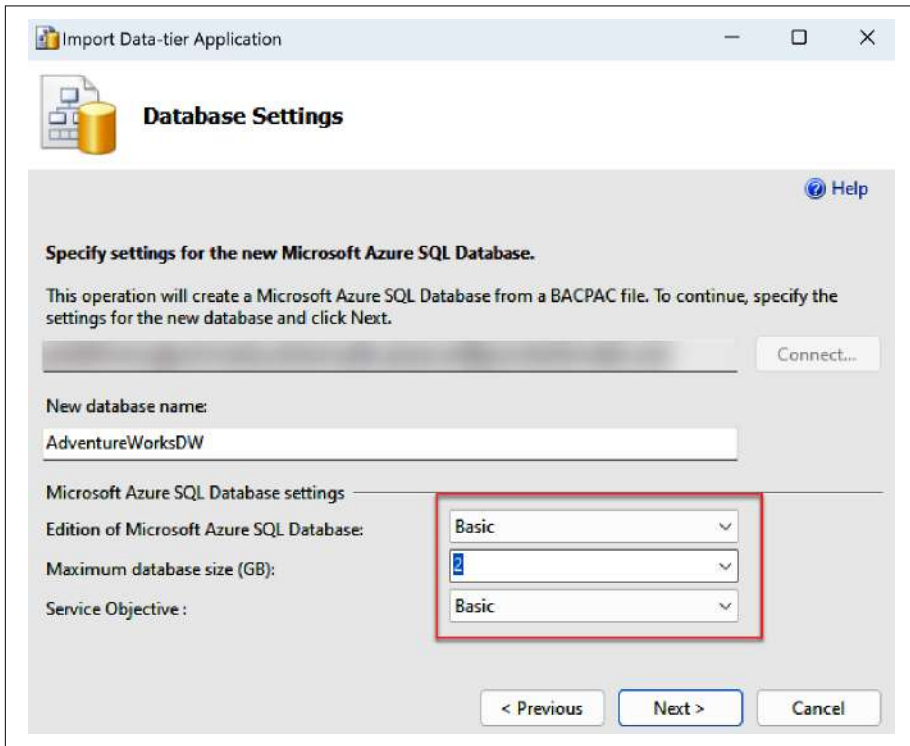


Figure P-12. Choose the smallest available database setting

If you choose to run the database *serverless*, you can further reduce costs, as the database (and all costs) will hibernate if you don't use it for one hour. Next time you access the database, it will wake up for you, which can lead to a timeout. Just reconnect again, and the connection will be established successfully.

Finally, run the script `001 Preparation.sql` when connected to database *AdventureWorksDW* to create all artifacts for the demo environment.

Additional Tools

The data community around Power BI is great. I really admire the smart people who combine a deep knowledge of the technology behind Power BI with understanding the needs of professional Power BI developers, and who also have the skill set to develop useful tools. Some of these people even develop their tools as open source and share them with the community. In this book, I refer to two such tools:

- Tabular Editor V2, the open source version of **Tabular Editor V3**, by Daniel Otykier (CTO at Tabular Editor ApS)
- **DAX Studio**, an open source tool by Darren Gosbell (senior program manager at Microsoft)

Demo Files

For every problem, I share one single Power BI Desktop file (*.pbix*) that contains the whole data model and all the solutions in DAX and in Power Query/M, as well as a connection to the tables in Azure SQL. This allows you to easily compare the different technical solutions with one another.

The majority of the examples have an educational purpose: for instance, in the file *Data multiple.pbix* you'll find the table `Order Date` three times with the exact same content. `Order Date` (DAX) is a version completely created in DAX, `Order Date` (PQ) is a version completely created in Power Query, and `Order Date` (SQL), as you might guess, is a version completely created in SQL and just loaded as is into Power BI. To avoid any misconceptions: in a practical scenario, it neither makes sense to create a table with the same content several times in your data model nor does it make sense to do some transformations in DAX while doing others in Power Query and/or SQL. Stick to one tool to make it easier to find any transformation.

You can find all files in the [GitHub repository for this book](#).

Now you're set for the first chapter, which will give you a basic understanding of what a data model is and what it consists of.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Shows for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.

For all DAX code (mostly in **Part III**), the following conventions also apply:

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

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

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



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at <http://github.com/MEhrenmueller/DataModeling>.

If you have a technical question or a problem using the code examples, please email bookquestions@oreilly.com.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Data Modeling with Microsoft Power BI* by Markus Ehrenmueller-Jensen (O'Reilly). Copyright 2024 Savory Data Gmbh, 978-1-098-14855-3.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

O'Reilly Online Learning



For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <https://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
<https://www.oreilly.com/about/contact.html>

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *<https://oreil.ly/DataModelingwithMicrosoftPowerBI>*.

For news and information about our books and courses, visit *<https://oreilly.com>*.

Find us on LinkedIn: *<https://linkedin.com/company/oreilly-media>*.

Watch us on YouTube: *<https://youtube.com/oreillymedia>*.

Acknowledgments

Writing a book looked like a lonely work in the beginning: writing down a concept and draft, going through my own material and doing research, developing and improving the examples, creating the screenshots and diagrams, transforming the thoughts and ideas into words, etc. But I soon found out that it is a collective endeavor; I was lucky enough to work with very engaged and motivated people who I want to bring in front of the curtain here.

Michelle Smith was the acquisition editor who challenged my book proposal and helped it become a useful draft to start from, and Shira Evans was the excellent development editor. I had countless hours of fruitful discussions with Shira in which she challenged the structure of the book, made sure I aligned with the publisher's guidelines, asked the right questions to make the book more readable, and kept me motivated during the whole process all the way through publication with her knowledge, experience, and happy mood. Thanks also to Katherine Tozer, Kate Dullea, Liz Wheeler, Alexis Browsh, and all the folks at WordCo Indexing Services for their work on the final pieces in production.

I am also very pleased to have convinced renowned top experts Shabnam Watson, Matt Allington, and Nikola Ilic to join in as technical reviewers for this project. I

appreciate the time they dedicated to this book. Their comments and findings made the book so much better.

Furthermore, I want to thank all my clients who keep me busy in projects and attendees of my seminars, workshops, and webinars through the past years. Solving their challenging problems, preparing materials for presentations, and answering their questions led finally to the idea of bringing everything together for this book.

Data Modeling 101

Chapters	Parts				
	Part 1 Data modeling in general	Part 2 Power BI	Part 3 DAX	Part 4 Power Query	Part 5 SQL
Understanding a data model	Chapter 1	Chapter 5	Chapter 9	Chapter 13	Chapter 17
Building a data model	Chapter 2	Chapter 6	Chapter 10	Chapter 14	Chapter 18
Real-world examples	Chapter 3	Chapter 7	Chapter 11	Chapter 15	Chapter 19
Performance tuning	Chapter 4	Chapter 8	Chapter 12	Chapter 16	Chapter 20

The goal of this first part of the book is to bring everybody onto the same page when it comes to the topic of data modeling. The chapters in this part are agnostic: the content, problems, and solutions are not specific to a particular type of database management system. You will be able to apply the knowledge you gain from this part to any relational or analytical database. This includes, of course, Power BI, Analysis Services tabular and Azure SQL DB, but is not limited to them, and you can apply all statements, information, and conclusions to database management systems from vendors other than Microsoft—to classical cubes, to data lakehouses, and so on.

These concepts have existed for decades and are so mature that I bet they'll be around for decades to come. Make sure to learn of all them so you'll understand why I insist on applying one transformation or another when it comes to data modeling in Power BI in the later parts of this book.

Chapter 1 introduces the following basic terms and concepts:

- Entities and tables
- Relations and their cardinality
- Primary and foreign keys

You'll learn how to combine information spread out into different tables with the help of *Set operators* and *join operators*, including problems you might face that, for example, could result in missing data or duplication of data. I discuss the three core possibilities of modeling data (combining everything into one single table, splitting the information in a way that avoids duplicates under all circumstances, and a compromise in the form of a so-called *dimensional model*). You'll learn to decide which to use under which conditions.

Based on the information in **Chapter 1**, **Chapter 2** teaches you how to transform the data model of your data source into a data model of the intended shape. This can be done via transformation steps, including the following:

- Normalizing and denormalizing tables
- Adding calculations
- Transforming flags and indicators into meaningful text
- Adding a dedicated table to contain all dates and/or timestamps
- Modeling dimensions when they can play different roles (e.g., a person can be in the role of an employee or in the role of a customer)
- Modeling dimensions when we need to track attribute changes over time
- Implementing hierarchies

In my experience, the challenges in **Chapter 2** are common in most data models. **Chapter 3** talks about rarer challenges, which you won't see in every data model. All of them are real-world problems I've had to tackle for my customers. To solve them, you'll need to combine different techniques, which make them more advanced.

Finally, in this part's performance tuning section, **Chapter 4**, I introduce what role a data model plays when it comes to guaranteeing a fast performance of the reports and queries based on the data model. Basically you need to decide whether you want to persist data in the shape you need it in for your analytics or only store the code for a query, which instead transforms the original data into the necessary shape on the fly.

What Is a Data Model?

This chapter covers the basics of data modeling, starting with basic terms, which will help establish the reasoning behind taking so much care of the data model. A data model that is optimized for creating reports and doing analysis is much easier to work with than one that is optimized for other purposes (e.g., 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.

The goal throughout this book is to create the data model as a star schema. By the end of this chapter, you'll know which characteristics of a star schema differentiate it from other modeling approaches. Each and every chapter reinforces why a star schema is so important when it comes to analytical databases in general and Power BI and Analysis Services tabular in particular. You will learn how to transform any data model into a star schema.

Transforming information of your data source(s) into a star schema is usually *not* an easy task. Quite the opposite; it can be difficult. 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, as well as those who are responsible for the data sources. You might face doubts (from others and yourself) about whether it's really worth all the effort instead of avoiding the struggle and pulling the data in as it is. At such a point, it's important to take a deep breath and evaluate whether a transformation would make the report creator's life easier. If so, then it's worth the effort. Repeat the data modeler's mantra with me: *make the report creator's life easier*.

Before I talk about transformations, I'll introduce some basic terms and concepts:

- What is a data model?
- What is an entity? What does an entity have to do with a table?

- Why should you care about relationships?
- Why do you need to identify different keys (primary, foreign, and surrogate) and understand the general meaning of cardinality?
- How can you combine tables (set operators and joins)?
- What are the data modeling options?

The first stop on our journey toward the star schema is learning what a data model is in general terms.

Data Model

A model is something that *represents* the real world. It does not replicate it. Think of a map. A map replicating the real world 1:1 would be impractical: it would cover the whole planet. Instead, maps scale down distances. Maps are created for special purposes. A hiking map contains (and omits) different types of information than a road map does, and a nautical chart looks completely different still. All of these are maps but with different purposes.

The same applies to a data model, which represents a certain business logic. As with a map, a data model will look different for different use cases. 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 identical business processes), as they will concentrate on different requirements. The challenges and solutions in this book will help you overcome obstacles when you build a data model for your organization.

Now for the bad news: there isn't one data model that rules them all. Also, it's impossible to create a useful data model with technical knowledge and no domain knowledge. But there is good news: this book will guide you through the technical knowledge necessary to successfully build data models for Power BI and/or Analysis Services tabular.

Don't forget to collect or record all requirements from the business before creating the data model. Here are some examples of requirements in natural language:

- We sell goods to customers and need to know the day and the SKU of the product we sold
- We need to analyze the 12-month rolling average of the sold quantity for each product
- Up to 10 employees form a project team and we need to report the working hours per project task

These requirements will help you determine what information you need to store in which combinations and at which level of detail. There might be more than one option for the design of a data model for a certain use case.

And, very importantly, you need to create the correct data model right from the beginning. As soon as the first reports are created on a data model, every change in the model bears the risk of breaking those reports. The later you discover inconsistencies and mistakes in your data model, the more expensive it will be to correct them. This cost hits everybody who created those reports—you yourself, but also other users who built reports based upon your data model.

The data model's design has a huge impact on the performance of your reports, which query 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 saving your organization money). From a different point of view, problems with the performance of a report or report creators who are unsure of which tables and columns to use to gain certain insights are a sure sign of a data model that can be improved by a better choice of design.

In “[Entity Relationship Diagrams](#)” on page 25, I describe graphical ways to document a data model's shape. But first, let's discuss the entities of a data model.

Basic Components

You need to understand a few key components of a data model before you dive in. In this section, I explain the basic parts of a data model. In “[Combining Tables](#)” on page 11, 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.

Remember that this part of the book isn't specific to Power BI. Some concepts may apply only when you prepare data before you connect Power Query to it (e.g., when writing a SQL statement in a relational database).

Entity

An *entity* is someone or something that 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 fictitious. And most have attributes: a name, value, category, point in time of creation, etc. These attributes are the information we are after when it comes to data. Attributes are displayed in reports to help the reader provide context, gain insights, and make decisions. They are used to filter displayed information to narrow down an analysis, too.

How do such entities make it into a data model? They're stored in tables.

Tables

Tables are the base of a data model. They have been part of data models since at least 1970, when Edgar F. Codd developed the relational data model for his employer, IBM. But collecting information as lists or tables was done way before the invention of computers, as you can see when looking at old books.

Tables host 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 isn't possible to store the place of birth in the column "birthday" for any row, for example). 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. You can see an example in [Table 1-1](#).

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.

Relationships

In most cases, relationships connect only two entities. In natural language, the relationship is represented by a verb (e.g., bought). Between the same two entities, more than one single relationship might exist. 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 relationships (ordered versus 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 are a typical example. Every employee (except maybe the CEO) needs to report to a supervisor. The reference to the boss is therefore an attribute. One column contains the identifier of the employee (e.g., [Employee ID]) and another contains the identifier of who this

employee reports to (e.g., [Manager ID]). The [Manager ID] in one row can be found as the [Employee ID] of another row in the same table.

Here are a few examples of relationships expressed in natural language:

- Dr. Smith *treats* Mr. Jones.
- Michael *attended* a data modeling course.
- Mr. Gates *owns* Microsoft.
- Mr. Nadella *is* the CEO.

When you start collecting the requirements for a report (and therefore for your data model) it makes sense to write them down as sentences in natural language, as in the preceding list. This is the first step. In “[Entity Relationship Diagrams](#)” on page 25, you will learn to draw tables and their relationships as entity relationship diagrams.

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-depth analysis:

- Dr. Smith treats Mr. Jones *for the flu*.
- Michael completed a data modeling course *with an A grade*.
- Mr. Gates owns *50% of* Microsoft.
- Mr. Nadella has been CEO *since February 4, 2014*.

You learned that entities are represented as rows in tables. The question that remains is how you can then connect rows with each other to represent relationships. The first step is to find a (combination of) columns that uniquely identify a row. Such a unique identifier is called a *primary key*.

Primary Keys

Both in the real world and in a data model, it’s important that you can uniquely identify a certain entity (a row in a table). People, for example, are identified via their names in the real world. When you know two people with the same (first) name, you might add something to their name (e.g., their last name) or invent a nickname (which is usually shorter than the first name and last name combined), so that you can make clear who you are referring to (without spending too much time). If somebody isn’t paying attention, they might end up referring to one person while another is referring to a different person (“Ah, you’re talking about the other John!”).

It's similar in a table: you can mark one column (or a combined set of columns, i.e., a *composite key*) as the primary key of the table. If you don't do that, you might end up with confusing reports because of duplicates (e.g., the combined sales of all Johns might be shown for every John).

The best practice is to use a single column as a primary key (as opposed to a composite key) for the same reason we use nicknames for people: it's shorter and, therefore, easier to use. You can only define one primary key.

Explicitly defining a primary key on a table has several consequences. It puts a unique constraint on the column(s) (which guarantees that no other row can have the same value(s) as the primary key and ensures the rejection of inserts and updates that would result in the violation of 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). 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 to avoid ending up with duplicate rows.

In a table, you can make sure that every row is uniquely identifiable by marking a row, or the combination of several rows, as the primary key. To make the example with [Employee ID] and [Manager 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), instead of using one of the columns of the source system as a primary key (e.g., first name and last name or social security number), you would introduce a new artificial ID, which only exists inside the data warehouse: a surrogate key.

Surrogate Keys

A *surrogate key* is an artificial value used only in the data warehouse or analytical database. It's 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 key column, which is independent of any source system. Think of it as a (bit weird and secret) nickname, which identifies an entity uniquely and will only be used to join tables but never to filter the content of a table. The surrogate key is never exposed to the report consumers and users of the analytical system.

Typically, the columns have "Key," "ID," "SID," etc. as part of their names (as in ProductKey, Customer_ID, and SID). The common relational database management systems are able to automatically find a value for this column for you. The 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 bits = $2^8 = 256$ values) and 8 bytes (= 8×8 bits = 64 bits = $2^{64} = 18,446,744,073,709,551,616$ values).

Sometimes global unique identifiers 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 (16 bytes) compared to an integer value (max. 8 bytes). 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 function independently from the source system (e.g., when an enterprise planning system [ERP] is changed, when the ERP system re-uses 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 *slowly changing dimension Type 2*, which I cover in [“Slowly Changing Dimensions” on page 49](#).

The next section will explain how to represent relationships of entities. The solution to the puzzle of how to relate entities to each other isn't 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 in the referencing table.

Foreign Keys

Foreign keys simply reference 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. The [Manger ID] column of the Employee table refers to column [Employee ID] in the very same table (Employee).

When you 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 referenced 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 referenced table.

The 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 explicitly add a row with -1 as it's primary key to the referenced table. This gives you 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 values so those rows are not lost in the result set; read more about “Joins” on page 13).

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.

How do you determine in which of the two entities involved in a relationship to store the *foreign key*? Before I can answer this question, we need to discuss different types of relationships.

Cardinality

The term *cardinality* has two meanings. It can be a number used to describe how many distinct values in one column (or combinations of values for several columns) you can find in a table. If you store a binary value in a column, e.g., either “yes” or “no,” then the cardinality of the column will be two.

The cardinality of the primary key of a table will always be identical to the number of rows in a table because every row of the table will have a different value. In a columnar database (like Power BI or Analysis Services tabular), the compression factor is dependent on the cardinality of a column (the fewer distinct values, the better the compression will be).

In the rest of the book, I mostly refer to the other meaning, which describes cardinality as how many rows can (maximally) be found in a related table for a given row in the referencing table. For two given tables, the cardinality can be any of the following:

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 to this other person.

*Many-to-many (m:m, *-*)*

For example, one employee works for many different projects. One project has many employees.

The cardinality is defined by the business rules. Maybe in your organization, a single order can be assigned to two customers simultaneously. Then the one-to-many assumption would be wrong, and you’d need to model this relationship as many-to-many. Finding the correct cardinalities is a crucial task when designing a data model. Make sure you fully understand the business rules to avoid incorrect assumptions.

If you want to be more specific, you can also describe whether 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). You might then change the relationship description from 1:1 to conditional:conditional (c:c) in your documentation.

Combining Tables

So far, you’ve learned that information (entities and their relationships) is stored in tables in a data model. Before I introduce rules, such as when to split information into different tables and when keep it together in on single table, I want to discuss how to combine information spread throughout different tables.

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 you 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 system you’re using must be able to (implicitly) convert to the data type of the column of the first query.

The first query sets both the data types and the names of the columns of the overall result. A set operator does not change the number or type of columns, only the number of rows. [Figure 1-1](#) illustrates the following explanation:

Union

Adds the rows from the second set to the rows of the first set. Depending on the database management system you are using, 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 that 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 that appear in both sets. Only rows from the first set that do not appear 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.

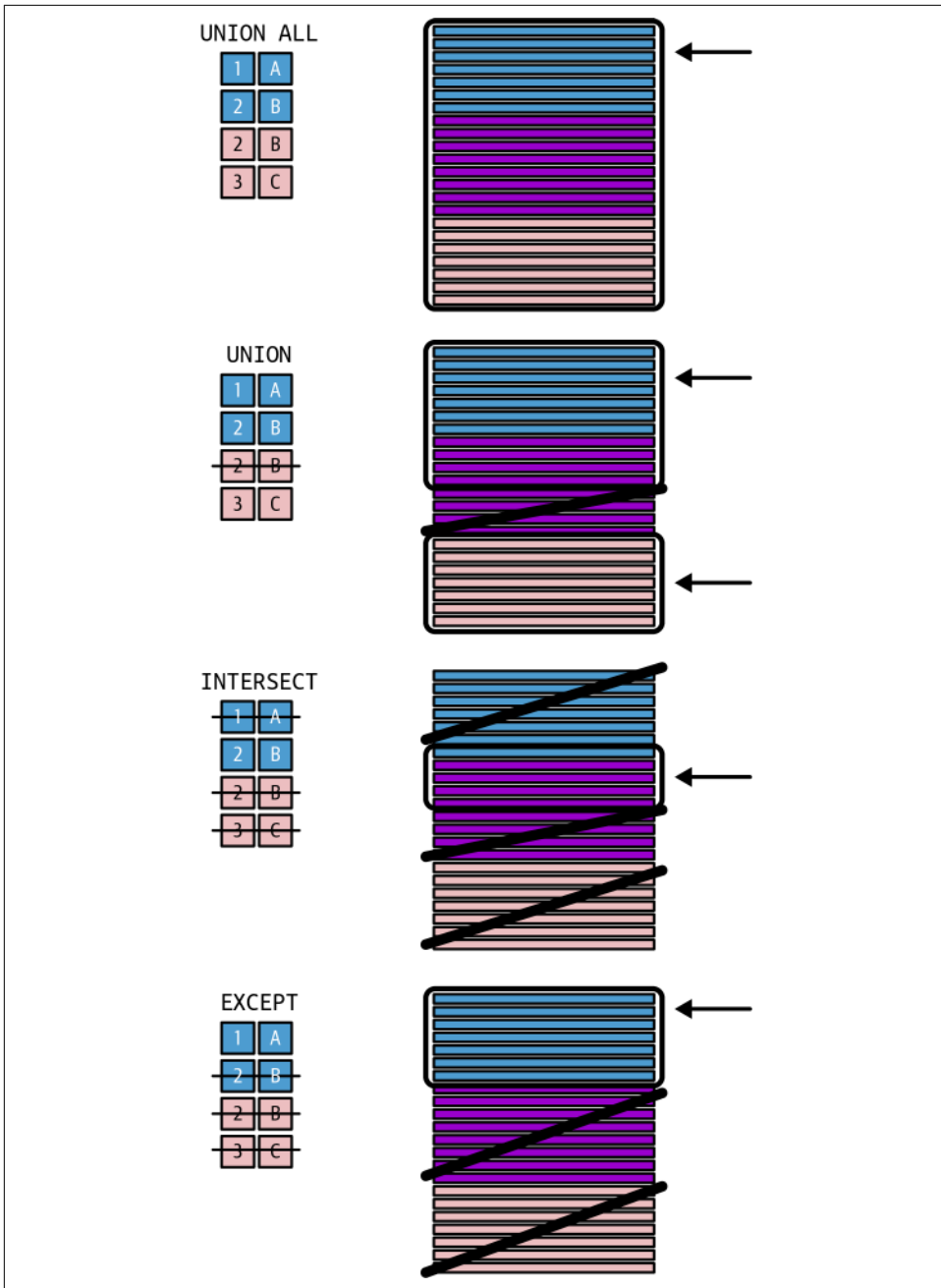


Figure 1-1. Set operators



To determine whether a row is identical or not, evaluate and compare the content of all columns of the row of the query. Pay attention here: while the primary keys listed in the result set might be identical, the names or description might differ. Rows with identical keys but different descriptions will not be recognized as identical by a set operator.

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

Joins

Joins are like set operators in the sense that they also combine two (or more) queries (or tables). Depending on the type of the join operator, you might end up with the same number of rows as the first table, more, or fewer 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, you'll usually 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).

Only in special cases would you compare other (non-key) columns with each other to join two tables. You will see examples of such joins in Chapters 7, 11, 15, and 19, where I demonstrate solutions to advanced problems. There, you'll also see examples for non-equi-joins. Non-equi-joins use operators like between, greater than or equal to, not equal, etc., to join the rows of two tables.

One example is about grouping values (binning). Binning is about finding the group a certain value falls into by joining the table containing the groups with a condition asking for values that are greater than or equal to the lower range of the bin and lower than the upper range of the bin. While the range of values form the composite primary key of the table containing the groups, the lookup value is not a foreign key: it's an arbitrary value, possibly 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 don't 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 (a very good idea in any case) to support these joins. If the primary key and foreign key columns have different names, a natural join will not work properly (e.g., when the primary key in the customer table is the column ID, while, in the order table, the foreign key is named CustomerID). The same is true in the opposite case, when columns in both tables have the same name but no relationship (e.g., both the Product table and the Product Category table have a Name column, which represents the name of the Product and the name of the Product Category, respectively, but they cannot meaningfully be used for the equi-join).



The important difference between set operators and joins is that joins add *columns* to the first table, while set operators add *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.

I imagine set operators as a combination of tables arranged vertically (one table underneath another) and join operators as a horizontal combination of tables (side-by-side). This metaphor isn't exact in all regards (set operators INTERSECT and EXCEPT remove rows, and joins also add or remove rows depending on the cardinality of the relationship or the join type) but it is, I think, a good starting point to differentiate them.

Earlier in this chapter, I used Employee as a 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*.

You can join two tables in the following ways:

Inner join

Looks for rows that 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 whom you can find orders. You can see a graphical representation in [Figure 1-2](#).

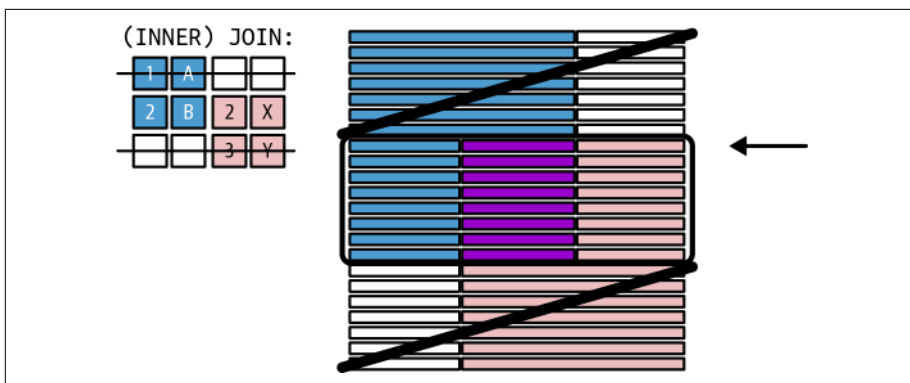


Figure 1-2. Inner join

This is similar to the INTERSECT set operator. But the result can contain the same, more, or fewer rows than the first table. It will contain the same number of rows, if for every row of the first table exactly one row in the second table exists (e.g., when every customer has placed exactly one 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 overcompensate for customers who didn't place an order). It will contain fewer 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).



There is a “danger” of inner joins: the result may skip some rows of one of the tables (e.g., the result will not list 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 [Figure 1-3](#).

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

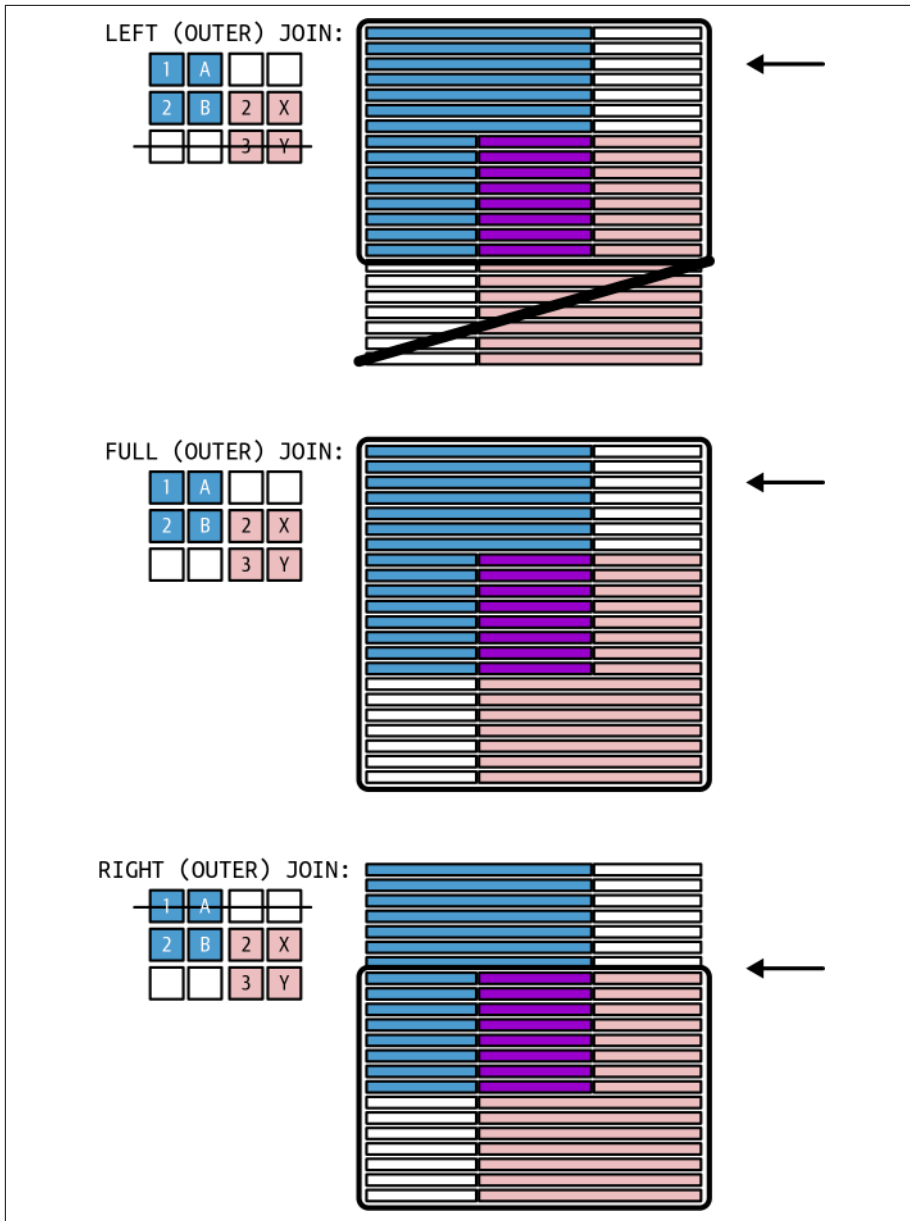


Figure 1-3. Outer join

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 you want null or 0 displayed as their order sales). To achieve this, you would select the rows from the Customer table and *left join* the Orders table to it. The first table (Customer) is considered the *left* table, and the joined table (Orders) is the *right* table in such a query.



This is easy to understand in SQL when you write all the tables in one single line, e.g., ... FROM Customer LEFT OUTER JOIN Order.... The Customer table is literally written to the left of the Order table, thus it is the *left* table. The Order table is literally right of the Customer table, thus it is the *right* table.

There is no similar set operator to achieve this. An outer join will have at least as many rows as an inner join. It's not possible that an outer join (with the identical join predicated) will return fewer 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 apply here for left, right, and full, as you can see in [Figure 1-4](#).

Anti-joins have a very practical use. For example, you want a list of customers who didn't order anything in the current year (to send them an offer they can't resist). There is no similar set operator to achieve this. The anti-join delivers the difference between an inner join and an outer join.

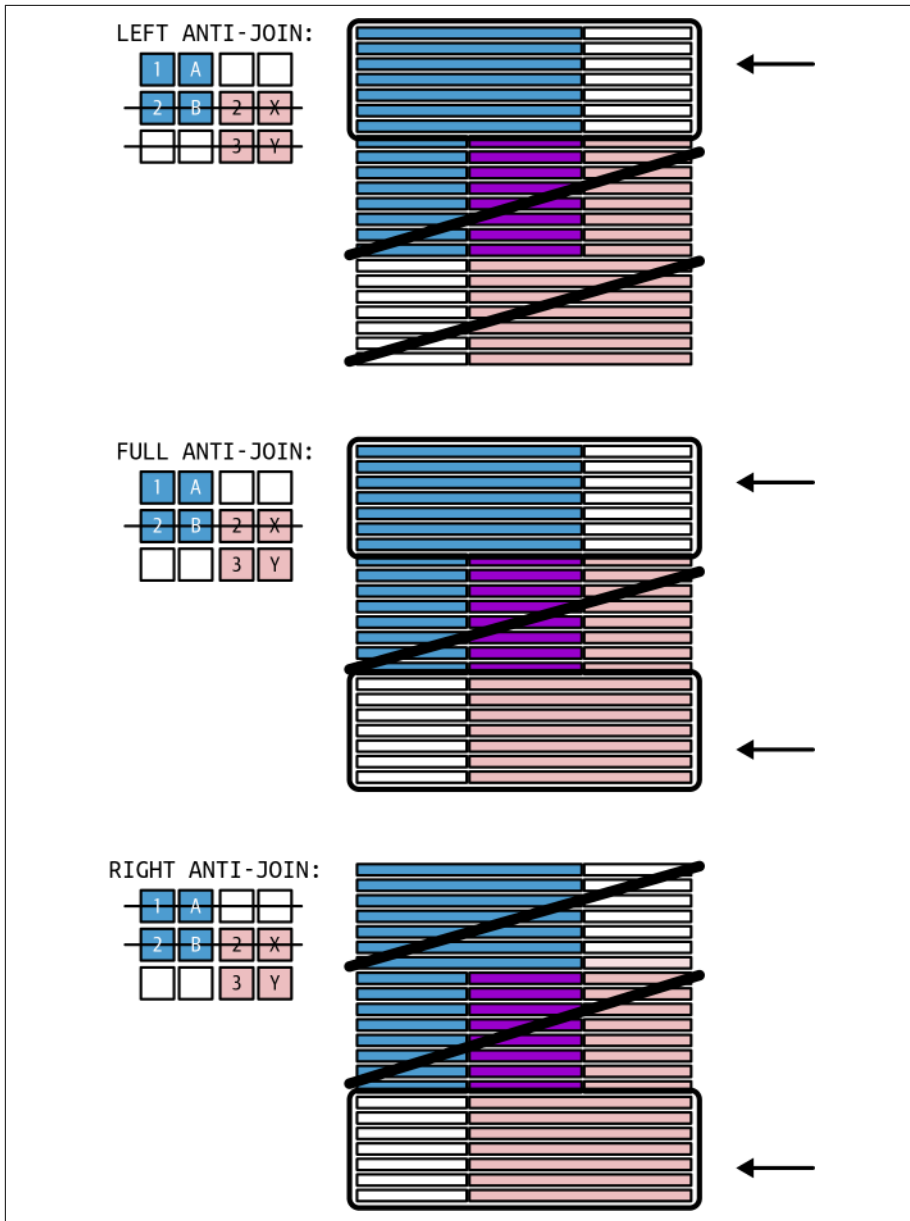


Figure 1-4. Anti-join

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 doesn't make sense (e.g., when combining every row of a sales table with every customer without considering whether the row of the sales table is for that customer or a different one).

Practically, you can create queries, which show possible combinations. For example, by applying a cross join on the sizes of clothes with all the colors, you get a list of all conceivable combinations of sizes and colors (independently of 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 explicitly point out combinations with no values available. You can see an example of the result of a cross join in [Figure 1-5](#).

CROSS JOIN:			
1	A	2	X
1	A	3	Y
2	B	2	X
2	B	3	Y

Figure 1-5. Cross join

Do you feel dizzy from all of the join options? Unfortunately, I need to add one more layer of complexity. As you've learned, when joining two tables, the number of rows in the result set might be smaller than, equal to, 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.

Join Path Problems

The Power BI data model has a fail-safe to avoid the join path problems described here. The problems can easily show up, however, when you combine tables in Power Query, in SQL, or your data source.

When you join the rows of one table to the rows of another table, you can face several problems, resulting in unwanted query results: loop, chasm trap, and fan trap. Let's take a closer look at them.

Loop

You face this problem in a data model if more than one single path exists between two tables. It doesn't 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. You're already talking about a loop when a data model is ambiguous. And this can exist not only 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 a due date, an order date, and a ship date column (Figure 1-6). All three date columns of the table FactResellerSales (DueDateKey, OrderDateKey, and SalesDateKey) have a relationship to the date column of the date table.

The tables contain the following rows (Tables 1-2 and 1-3).

Table 1-2. DimDate

DateKey
2023-08-01
2023-08-02
2023-08-03

Table 1-3. FactResellerSales

DueDateKey	OrderDateKey	ShipDateKey	SalesAmount
2023-08-01	2023-08-01	2023-08-02	10
2023-08-01	2023-08-02	2023-08-02	20
2023-08-01	2023-08-02	2023-08-03	30
2023-08-03	2023-08-03	2023-08-03	40

If you join the DimDate table with the FactResellerSales simultaneously on all three DateKey columns by writing a join predicate like this:

```
DimDate.DateKey = FactResellerSales.DueDateKey AND DimDate.DateKey =  
FactResellerSales.OrderDateKey AND DimDate.DateKey =  
FactResellerSales.ShipDateKey
```

the result would show only a single row (namely the row that—by chance—was due, ordered, and shipped on the same day, 2023-08-03; shown in Table 1-4). We might safely assume that many business orders are not due or shipped on the day of the order. Such sales rows would not be part of the result. This might be an unexpected behavior, returning too few rows.

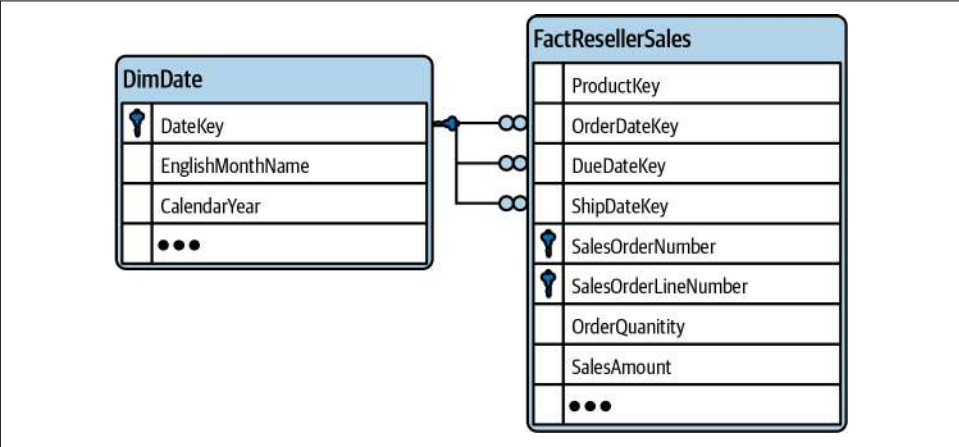


Figure 1-6. Join path problem: loop

Table 1-4. Query result

DateKey	DueDateKey	OrderDateKey	ShipDateKey	SalesAmount
2023-08-03	2023-08-03	2023-08-03	2023-08-03	40

The solution for a loop is (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 describes a situation in a data model wherein you have a converging many-to-one-to-many relationship (see Figure 1-7). 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 (see Tables 1-6 and 1-7). Both tables can be filtered over a common table, let's say a date table (Table 1-5). 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.

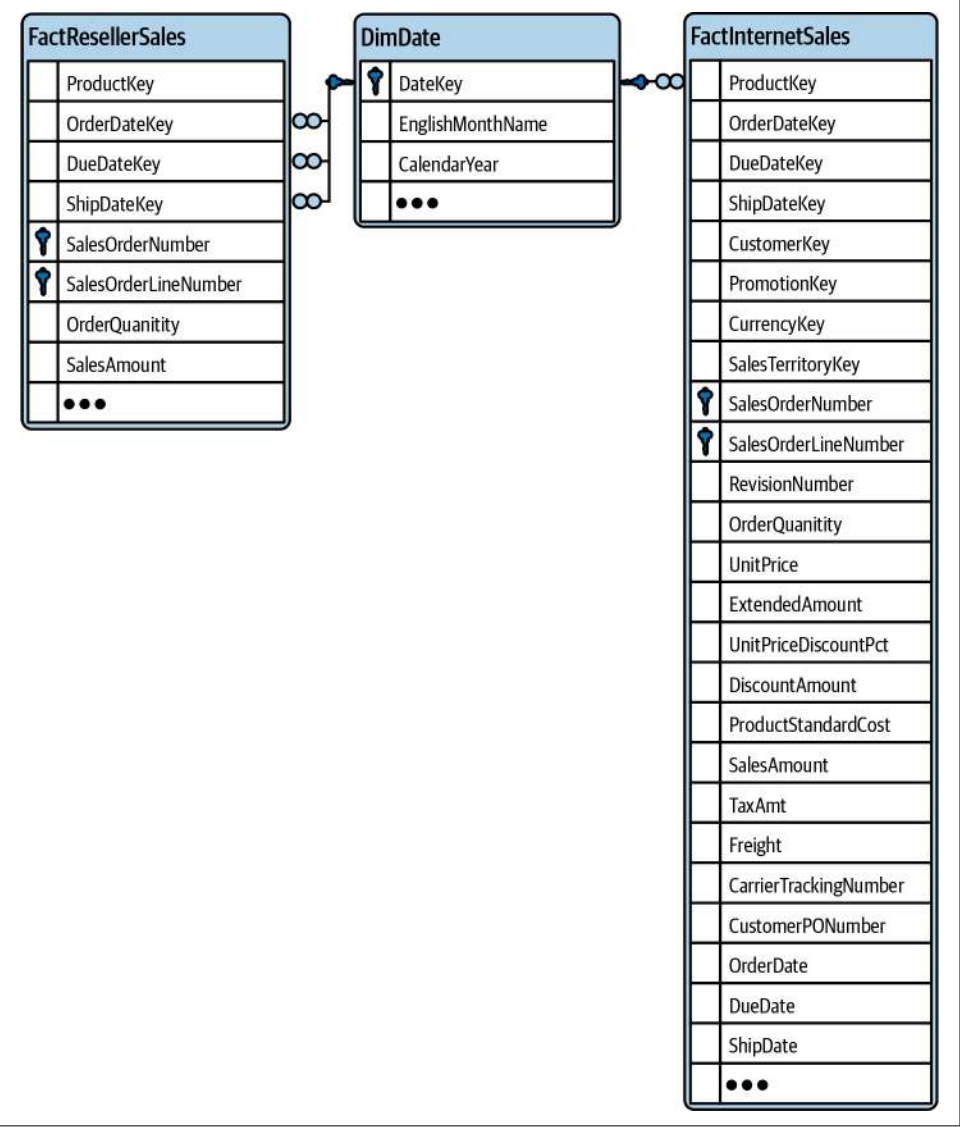


Figure 1-7. Join path problem: chasm trap

Table 1-5. DimDate

DateKey
2023-08-01
2023-08-02
2023-08-03

Table 1-6. FactResellerSales

OrderDateKey	SalesAmount
2023-08-01	10
2023-08-02	20
2023-08-02	30
2023-08-03	40

Table 1-7. FactInternetSales

OrderDateKey	SalesAmount
2023-08-01	100
2023-08-02	200
2023-08-03	300

Joining DimDate and FactResellerSales on DimDate.OrderDateKey = FactResellerSales.OrderDateKey would result in four rows, where the 2023-08-02 row of DateKey will be duplicated (due to the fact that on this day there are two reseller sales). So far so good. The (chasm trap) problem comes when you join the FactInternetSales table to this result (on DimDate.OrderDateKey = FactResellerSales.OrderDateKey). As the result of the previous join duplicating the 2023-08-02 row of DimDate, the second join will also duplicate all rows of FactInternetSales for this day. In the example, the row with SalesAmount 200 will appear twice in the result. If you add the numbers up, you will incorrectly report an internet sales amount of 400 for 2023-08-02 in the combined query result. This problem appears independently of using an inner or outer join. (In Table 1-8, I abbreviate FactResellerSales as FRS and FactInternetSales as FIS.)

Table 1-8. Query result

DateKey	FRS.OrderDateKey	FRS.SalesAmount	FIS.OrderDateKey	FIS.SalesAmount
2023-08-01	2023-08-01	10	2023-08-01	100
2023-08-02	2023-08-02	20	2023-08-02	200
2023-08-02	2023-08-02	30	2023-08-02	200
2023-08-03	2023-08-03	40	2023-08-03	300

The solution for 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 Power Query/M and SQL.

Fan trap

You can step into a fan trap 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 (see [Figure 1-8](#)). For example, you could store the freight cost in a sales header table that holds information per order. When you join this table with the sales detail table that 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.

Tables [1-9](#) and [1-10](#) exemplify this:

Table 1-9. SalesOrderHeader

SalesOrderID	Freight
1	100
2	200
3	300

Table 1-10. SalesOrderDetail

SalesOrderID	SalesOrderLineID	OrderQty
1	1	10
1	2	20
2	1	30
3	1	40

Joining the tables `SalesOrderHeader` and `SalesOrderDetail` on the `SalesOrderID` leads to duplicated rows of the `SalesOrderHeader` table; the “1” row for `SalesOrderID` has two order details and will be duplicated. When you naively sum up the `Freight`, you would falsely report 200, instead of the correct number, 100.

Table 1-11. Query result

SalesOrderID	Freight	SalesOrderLineID	OrderQty
1	100	1	10
1	100	2	20
2	200	1	30
3	300	1	40

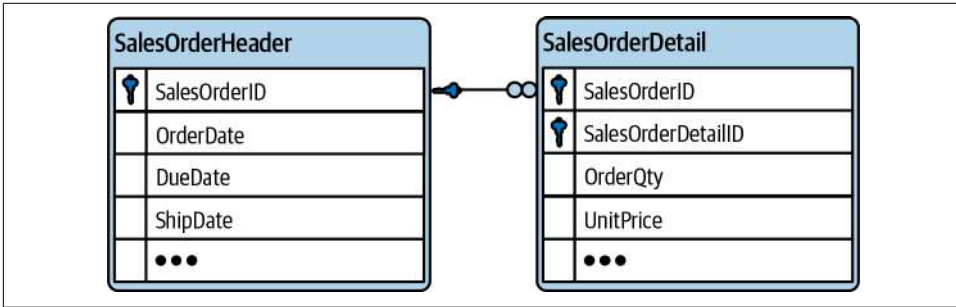


Figure 1-8. Join path problem: fan trap

Similar to the chasm problem, the solution for the fan 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.

As you can see 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

An entity relationship diagram (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 ([Lucidchart](#) has a nice overview about the most common notations).

In my opinion, it’s not so important which notation you use—it’s more important to have an ERD on 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, or sub-ERDs.

Deciding on the cardinality of a relationship and documenting it (e.g., in the form of an ERD) will help you find out in which table you need to create the foreign key. This section explores examples of tables of different cardinality.

The cardinality of the relationship between customers and their orders should be a one-to-many relationship. One customer could 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 only one customer. This knowledge helps you decide if we need to create a foreign key in the customer table to refer to 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 correctly reflect the reality. That's why you need a [Customer Key] (as a foreign key) in the order table instead, as shown in [Figure 1-9](#). Then, every row in the order table can only reference a single customer, and a customer can be referenced by many orders.

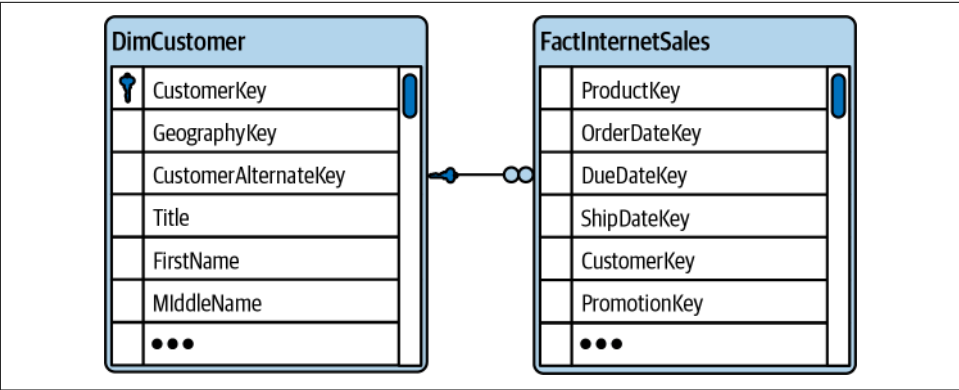


Figure 1-9. ERD for customer and order tables

In a case where an order could be associated with more than a single customer, you 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 collect the reasons for sales from your customers. The same reason will be given for more than one sale, and a customer might give you several reasons for why they made the purchase.

Typically, you 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 1-10](#)) 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's primary key) and a many-to-one relationship to the sales reason table (over the sales reason table's primary key). It's called a *bridge table* because it bridges the many-to-many relationship between the two tables and converting it into two one-to-many relationships.

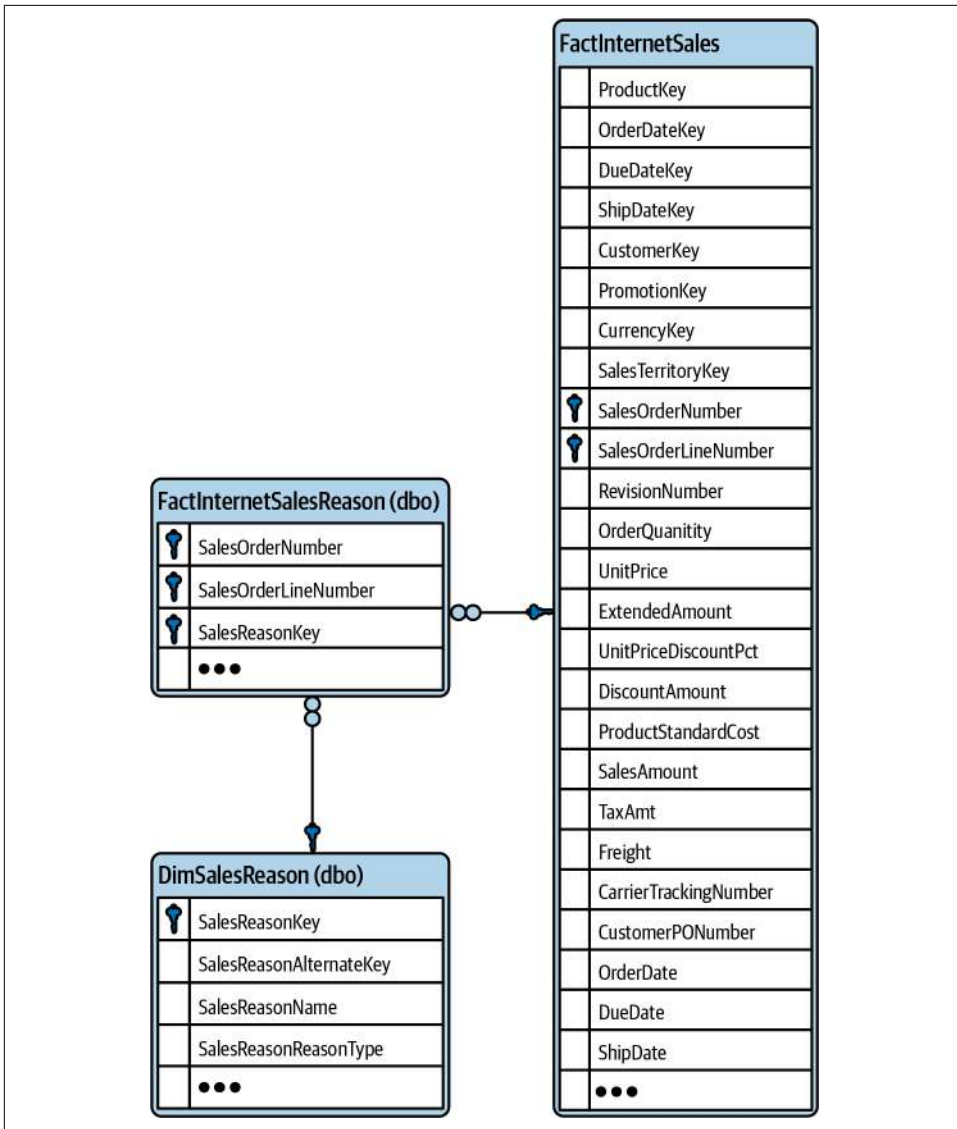


Figure 1-10. ERD for sales and sales reason tables

In later parts of this book, you will learn practical ways to create ERDs 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's about time we talk about different options for spreading information over tables and relationships in a data model. That's what the next sections will teach you.

Types of Tables

Basically, you can assign each table in your data model to one 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

Lookup tables are used to store more detailed information that you don't want to repeat in every row of the entity table. These tables are also referred to as master, main data, or dimension tables. Examples include customer, product, etc.

Bridge table

A bridge table changes a single many-to-many relationship into 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, you might link a table containing all employees and a table containing all projects.

Maybe you don't want to split your data into tables but want to keep it in one table. In the next section, I'll 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's easily read by humans; 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 table (or more) and list all relevant information as columns per a 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 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. A proper data model needs always to consist of more

than one table, if you don't want to encounter difficulties (as “[A Single Table to Store It All](#)” on page 103 points out). In the next section, you will learn rules for splitting columns from one table into several tables to achieve the goal of a redundancy-free data model.

Normal Forms

Codd, the inventor of relational databases, introduced the term *normalizing* in the context of databases. Personally, I don't like the term much. I think it's confusing; it's hard to tell what's normal and what's not, if you think about life in general and databases in particular. But I like the idea and the concept behind this term: the ultimate goal of normalizing a database is to remove redundancy, which is a good idea in many situations.

If you wanted to store the name, address, email, phone, etc. of the customer for each and every order, you could store this information in a redundant manner, requiring you to use more storage space than necessary, due to the duplicated information. Such a solution also makes editing the data overly 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 your system but didn't order anything yet, you have to store a placeholder value in the columns that contain the order information until an order is placed. If you delete an order, you have to pay attention, so as not to accidentally also remove the customer information, in case this was the customer's only order in the table.

To normalize a database, you apply a set of rules to bring it from one state to the other. Here are the rules to bring a database into the *third normal form*, which is the most common normal form:¹

Rule of the first normal form (1NF)

You need to define a primary key and remove repeating column values.

Rule of the second normal form (2NF)

Non-key columns are fully dependent on the primary key.

Rule of the third normal form (3NF)

All normal form are directly dependent on the primary key.

¹ If you want to dig deeper, you will find books explaining the *Boyce-Codd*, *fourth* and *fifth* normal forms, which I consider mainly of academic interest, and less practically relevant.



The following sentence has helped me 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* (origin unknown).

Let's apply those rules on a concrete example in [Table 1-12](#).

Table 1-12. A table violating the rules of normalization

StudentNr	Mentor	MentorRoom	Course1	Course2	Course3
1022	Jones	412	101-07	143-01	159-02
4123	Smith	216	201-01	211-02	214-01

This is one single table containing all the necessary information for our hypothetical scenario. In some situations, such a table is very useful, as laid out in “[A Single Table to Store It All](#)” on [page 28](#). But this is not a good data model, when it comes to Power BI, and it clearly violates all three rules of normalization (you need to define a primary key and remove repeating column values; non-key columns are fully dependent on the primary key; all attributes are directly dependent on the primary key).

In this example, you see repeating columns for the courses a student attends (columns `Course1`, `Course2`, and `Course3`). Such a schema limits the amount of courses to three, and creating a report on how many students visited a certain course is over-complicated, as you need to look in three different columns. Sometimes information is not split into several columns but all information is stored in a single column, separated by commas, or stored in a JSON or XML format (e.g., a list of phone numbers). Again, querying will be extra hard, as the format of the input cannot be forced. Some might delimit the list using a comma, others might use a semicolon, etc. These examples violate the rule of the first normal form, as well. You need to deserialize the information, and split the information into rows instead, so that you get just one column with the content split out into separate rows. This transforms the table toward 1NF.

Here, somebody might accidentally assign a student to a given course more than once. The database could not prohibit such a mistake. Yes, you could add a check constraint to enforce that the three columns must have different content. But somebody could add a second row for student number 1022, and add course 143-01.

Here, the definition of a primary key comes into play. A primary key uniquely identifies every row of this new table. In this first step, I don't introduce a new (surrogate) key but can live with a composite primary key. In [Table 1-13](#), the column headers, which make up the primary key, are printed underlined (StudentNr and Course).

Table 1-13. A table in first normal form (with a composite primary key consisting of StudentNr and Course)

<u>StudentNr</u>	<u>Mentor</u>	<u>MentorRoom</u>	<u>Course</u>
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 2NF, start at a table in 1NF and guarantee that all columns are *functionally dependent* on (all columns of) the primary key. A column is functionally 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 at the table makes it clear that the column Mentor is functionally 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 you can safely state that columns Mentor and MentorRoom are functionally dependent on only the StudentNr, but not on Course. Therefore, the current design violates the rules for 2NF.

Keeping it like this would allow you to introduce rows with the same student number, but different mentors or mentor rooms, which is not possible from a business logic perspective.

To achieve the 2NF, you have to split the table into two tables. One should contain columns StudentNr, Mentor, and MentorRoom (with StudentNr as its single primary key) (see Table 1-14). A second one should contain StudentNr and Course only (see Table 1-15). Both columns form the primary key of this table.

Table 1-14. Student table in 2NF (with primary key StudentNr)

<u>StudentNr</u>	<u>Mentor</u>	<u>MentorRoom</u>
1022	Jones	412
4123	Smith	216

Table 1-15. StudentCourse table in 2NF (with a composite primary key consisting of StudentNr and Course)

<u>StudentNr</u>	<u>Course</u>
1022	101-07
1022	143-01

<u>StudentNr</u>	<u>Course</u>
1022	159-02
4123	201-01
4123	211-02
4123	214-01

The rules for 3NF require that there is no functional dependency on non-key columns. In our example, the column `MentorRoom` is functionally dependent on the 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 of the mentee. In the current version of the data model, it would be possible to insert rows with wrong combinations of mentor and mentor room.

Therefore, you have to split the data model into three tables (Tables 1-16, 1-17, and 1-18), 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 1-16. Student table in 3NF (with primary key `StudentNr`)

<u>StudentNr</u>	<u>Mentor</u>
1022	Jones
4123	Smith

Table 1-17. Mentor table in 2NF (with primary key `Mentor`)

<u>Mentor</u>	<u>MentorRoom</u>
Jones	412
Smith	216

Table 1-18. The StudentCourse table is in 3NF as well (with a composite primary key consisting of `StudentNr` and `Course`)

<u>StudentNr</u>	<u>Course</u>
1022	101-07
1022	143-01
1022	159-02
4123	201-01
4123	211-02
4123	214-01

Table 1-18 is free of any redundancy. Every single piece of information is only stored once. No anomalies or violations to the business logic can happen. This is a perfect data model to store information collected by an application.

The data model is, however, 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). These characteristics make this data model less than ideal for analytical purposes. Therefore, I will introduce you to dimensional modeling in the next section.

Dimensional Modeling

Data models in 3NF (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 only has to happen in one place (content of a single column in a single row in on a single table).

Unfortunately, normalized data models are hard to understand. If you look at the ERD of a model for even a 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 model. If a data model is hard to understand for IT folks, how hard will it then be for domain experts to understand?

Such data models are also hard to query. In the process of normalizing, multiple tables get created, so querying the information in a normalized data model requires you to join multiple tables together. Joining tables is expensive. It requires a lengthy query to be written (and the lengthier, the higher the chance for making mistakes; if you don't believe me, reread [“Joins” on page 13](#) and [“Join Path Problems” on page 20](#)), and it requires you to physically join the information spread out from different tables by the database management system. The more joins, the slower the query.

Therefore, let's discuss 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 little as I like the term *normalized*, I dislike the term *denormalized* even more. Denormalizing can be easily misunderstood as the process to fully reverse all steps done during *normalizing*. That's wrong. A dimensional model reintroduces redundancy for some tables, 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 that's easy for the report creators to understand and use and allows for fast query performance. A dimensional model is common for data warehouses (DWHs), and Online Analytical Processing (OLAP)

systems—also called cubes—and is the optimal model for Power BI and Analysis Services tabular.

In a dimensional model, most of the attributes (or tables) can be either seen as a *dimension* (hence the name dimensional modeling) or as a *fact*.²

A *dimension table* contains answers to How? What? When? Where? Who? Why? The answers are used to filter and group information in a report. This kind of table can and will be wide (it can contain loads of columns). Compared to fact tables, dimension tables will be 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, but are fully denormalized. Think of the number of articles (dimension) a retailer sells, compared to the number of sales transactions (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 make sure you keep fact tables narrow (add columns only if really necessary), because compared to dimension tables, fact tables can be relatively big in terms of the number of rows (“long”). You want fact tables to be fully normalized.

Fact tables are on the “many” side of a relationship. If there isn’t a special reason, then a fact table won’t contain a primary key because a fact table is not—and never should be—referenced by another table. Every bit you save in each row adds 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 an additive, semiadditive, or nonadditive nature. 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 what Power BI (and a *columnstore index* in Microsoft’s relational databases) is optimized for. For certain reasons, you might keep a dimension table (partly) normalized and split information over more than one table. Then, some of the dimension tables will contain a foreign key. A star schema is preferred over a snowflake schema because in comparison, a snowflake schema:

² Later I’ll introduce you to other types of tables as well.

- Has more tables (due to normalization)
- Takes longer to load (because of the bigger amount of tables)
- Makes filtering slower (due to necessary additional joins)
- Makes the model less intuitive (instead of having all information for a single entity in a single table)
- Impedes the creation of hierarchies (in Power BI/Analysis Services tabular)

Of course, a dimension table may contain redundant data, due to denormalizing. In a data warehouse scenario, this isn't a big issue; there's only one process that can add and change rows to the dimension table (see **“Extract, Transform, Load” on page 36**).

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 refers to the level of detail of a table. On the one hand, you 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, you can also look on the granularity in the following terms:

Transactional fact

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

Aggregated fact

In an aggregated fact table, some foreign keys might be left out, or you can use a foreign key to a dimension table of different granularity. The fact table might contain sales per day (and reference a “day” dimension table). The aggregated fact table sums up the sales per month (and references a “month” dimension table). Or, you can pick a placeholder value for the existing foreign key (e.g., the first day of the month of the dimension table on the day level) 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 related to a 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 don't reduce the number of foreign keys but reduce the granularity of the foreign key on the date table, 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 to create a periodic snapshot on the month level. This is common with stock levels and other measures from a balance sheet. Queries are much faster when you have the correct number of available products in stock per day or month, instead of adding up the initial stock and all transactions until the point in time you need to report.

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 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 a table's granularity stays constant for all rows of a table. For example, you should not store aggregated sales per day in a table that is already at the granularity level 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 product group in a fact table, it is advised to have a dimension table with the product group as the primary key.

Now that you know how the data model should look, 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*, introduced in the next section. In later chapters, I'll give concrete tips, tricks, and scripts for using Power BI, DAX, Power Query, and SQL to implement transformations.

Extract, Transform, Load

By now, I hope I've made clear that a data model that is optimized for an application can look very different from a data model for the same data that 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, and sometimes you can query tables in the application's database.

Transform

Transforming the source data starts with easy tasks such as giving tables and columns user-friendly names (nobody wants to see, say, column EK4711 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 “Building a Data Model” sections of each part, you'll learn concepts and techniques to achieve this.

Load

Because 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, storing extracted and transformed data so that it can be queried quickly and easily is recommended (e.g., in a Power BI semantic model 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 compared to tasks in a restaurant kitchen. The cooks have dedicated tools to process the food and use all their knowledge and skills to make the food both good-looking and tasty, when served on a plate to the restaurant's customer. It's a great analogy for what happens during ETL because we use tools, knowledge, and skills to transform raw data into savory data that encourages an appetite for insights (hence the name of my company). Such data can then easily be consumed to create reports and dashboards.

Because the challenge of extracting, transforming, and loading data from one system to another is widespread, plenty of tools are 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's straightforward to adopt the job in case the table changes. These jobs are then put into the correct order by one additional orchestration job.

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. The order is mostly determined by which tool you are using (if you need or ought to first persist data before you transform it, or if you can transform it “on-the-fly” when loading the data). The most important thing is that the final result of the whole

process must be accessible easily and quickly 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, as is applying 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 and share the transformations as Power BI dataflow or use Power Query (inside Power BI Desktop) and share the result as a Power BI semantic model.

Only implement the transformations in the report layer as a last resort (better to implement it there instead of not implementing it at all). The “earlier” in your architecture you implement the transformation, the more tools can be employed, and the more accessible your product will be for users (like data engineers, data scientists, analysts, etc.). Something implemented only in the report is only available to the users of the report. If you need the same logic in another report, you need to re-create the transformation there (and face all consequences of code duplication, like a 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 semantic model or Analysis Services tabular database benefit from them. When you 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 that can hold all the necessary data and allows for your transformations), then a more widespread population of your organization will have access to clean and transformed information, without transformations needing to be repeated. You can connect Power BI to those tables and not need to apply any transformations.

Every concept introduced so far is based on the great work of two giants of data warehousing: Ralph Kimball and Bill Inmon.

Ralph Kimball and Bill Inmon

A book about data modeling would not be complete without mentioning (and referencing) 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. Regarding their differences, they never found compromises, and they “fought” about them (and against each other) in their articles and books.

For both, dimensional modeling (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 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. The data marts are shaped in a star schema fashion. Even when “Agile project management” wasn’t (yet) a thing, when Kimball described his concept, they clearly matched easily. Concentrating on smaller problems and creating data marts for them allows for quick wins. Of course, there is a risk that you won’t 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.

Kimball invented the concept of an *Enterprise Data Bus* to make all dimensions conform. He retired in 2015, but you can find useful information at [The Kimball Group](#), and his books are still worth a read. Their references and examples to SQL are still valid. He didn’t mention Power BI or the Analysis Services tabular model, which were only emerging then.

On the other hand, Inmon favors a top-down approach: you need to create a consistent data warehouse in first place. He called this central database 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 into a star schema). 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 the Data Warehouse*, 4th Ed. (Wiley, 2005) and are worth read as well. Inmon also supports the Data Vault modeling approach (“[Data Vaults and Other Anti-Patterns](#)” on [page 40](#)) and is an active publisher of books around data lake architecture.

If you want to dig deeper into the concept of a star schema (which you should!) I strongly recommend reading Chris Adamson’s masterpiece *Star Schema: The Complete Reference* (McGraw Hill, 2010).

Over the years, many different data modeling concepts have been developed and many different tools to build reports and support ad hoc analysis have been created. In the next section, I 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 instead.

Data Vaults and Other Anti-Patterns

I won't go into many details of how you can implement a Data Vault architecture. It is, however, important to lay out that a Data Vault is merely a data modeling approach that makes your ETL process flexible and robust against changes in the structure of the data source. The Data Vault's philosophy is to postpone cleaning of data to when it reaches the business layer. As easy as this approach makes the lives of data warehouse/ETL developers is proportional to how difficult it will make the lives of the business users. Remember: this book aims to describe how you can create a data model that makes the end user's life easier.

A Data Vault model is somewhere between a 3NF 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. There is a problem, 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 were really slow). You will sooner or later suffer from overcomplicated DAX calculations too.

That's why I strongly recommend that you *not* use any of the following data model approaches for Power BI and Analysis Services tabular:

Single table

See my reasoning in “[A Single Table to Store It All](#)” on page 28.

A table for every source file

This is a trap non-IT users easily step into. A table should contain attributes of only one entity. Often, a flat file or an Excel spreadsheet contains a report and not information limited to one entity. Chances are high that when you create a data model with one table per file, the same information is spread out over different tables, and that many of your relationships will 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 for querying. The number of tables and necessary joins makes it hard to use and impairs query response times. Chances are high that query performance is less than optimal, and that you will suffer from join path problems (see “[Join Path Problems](#)” on page 20).

Header—detail

Separating things like the order information and the order line information into two tables requires you 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 just one 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 at 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 I describe these as anti-patterns is not that these modeling approaches, from an objective point of view, are worse than star schema. The only reason is that many reporting tools benefit from a star schema so much that it is worthwhile 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 Power Pivot and SQL Server’s columnstore index) is fully optimized for star schemas 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 about how hard it is to write a letter in Excel or that many features to do your table calculations are missing in Word. Your mindset toward Power BI should be similar: you can use any data model in Power BI, but you should not start complaining about the product unless you have your data modeled 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 coming up in the rest of the book. Here is a short refresher of what you've learned so far:

- The basic parts of a data model: tables, columns, relationships, primary keys, and foreign keys.
- 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, and that's why they are the preferred data model for application databases. Dimensional modeling re-introduces some redundancy to make them easier to understand and to allow for faster queries (because fewer joins are necessary).
- Transforming of the data model (and much more) during the ETL process, which extracts, transforms, and loads data from data sources into the data warehouse.
- A rough overview about the contrary ideas of the two godfathers of data warehouses, Ralph Kimball and Bill Inmon.
- Why it is so important to stick to a star schema when it comes to Power BI and Analysis Services tabular.

Building a Data Model

Traditionally, we speak of Online Transactional Processing (OLTP) databases on one hand and Online Analytical Processing (OLAP) databases on the other. The term *online* isn't related to the internet here; it 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 if you're my age (or older); asynchronous queries were commonly used until the 1990s (and might still exist in main-frame computers). *Transactional* means that the purpose of the database is to store real-world events (transactions). This is typical for databases behind any application you can think of, such as the software your bank uses to track the movement of money or the retailer that keeps track of your orders and delivery. Databases for such use cases should avoid redundancy under all circumstances. A change of your name should not require updating only in a single place, rather than necessitating a complicated query to persist the new name through several tables in the database.

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, 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 data for the sole purpose of making analytical queries and reports easy, convenient, and fast (*make the report creator's life easier*).

Building an analytical database (and transforming data from data sources that were built with other goals in mind into a dimensional model) can be a challenge. This chapter will help you understand those challenges and how to overcome them.

As “[Dimensional Modeling](#)” on page 33 mentions, you need to normalize the fact tables and denormalize the dimension tables; more on this in the next section. You should also add calculations and transform flags and indicators into meaningful information to make the data model ready to use for reports. I recommend building a

dedicated date (and maybe an additional time) dimension so that the report creator doesn't 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'll 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 “[Dimensional Modeling](#)” on page 33: *normalizing* and *denormalizing* are terms to describe removing or adding 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 hand, should contain redundant information, where appropriate. In the next two sections, you will learn where to avoid redundancy in an analytical database as well, and where to make information explicitly redundant.

Normalizing

Normalizing means applying rules to the data model with the ultimate goal of avoiding redundancy. In “[Dimensional Modeling](#)” on page 33, you learned about the importance of normalizing a data model and why this is so important for OLTP databases. Normalizing is important for every table in an OLTP, but should be only done for specific tables (fact tables) in an analytical data model.

Normalizing is also necessary for fact tables in a dimensional model. As we've discussed, fact tables are the biggest tables in a data warehouse in terms of numbers of rows, and more rows constantly get 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 welcome. Think about this: if you save a single byte per row in a table containing one million rows, you save one megabyte of data. If you save 10 bytes in a table containing one billion rows, you save 10 gigabytes of data for this table. Reducing data will lessen pressure on the given infrastructure. And scanning less data will also lead to faster reports.

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; you don't want to have any tables in the model that 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 that contains information not unique to the primary key of the table.

When you model a natural hierarchy within one table, you will face redundancy. For example, one product category can consist of more than one 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 in addition 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 undesirable and should be avoided under all circumstances. In an analytical database, this type of redundancy is desirable and recommended for several reasons:

- Having all information about an entity in one 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 used 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 ameliorated, as the storage engine's automatic compression algorithm is optimized for such scenarios.) Therefore, the drawback 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).

Calculations

It's a good idea to add calculations 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 the cases of Power BI and Analysis Services, this means in the DAX language):

Additive

Many calculations can be calculated on top of results of finer 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. Stock levels are a typical example. 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 3 kg in one freezer and another 2 kg in a second freezer. But it does not make sense to add up the individual stock level for different days: if we had 5 kg yesterday and today only 1 kg is left, then adding these two numbers up to 6 kg 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 time frame.

Non-additive

Some calculations cannot be aggregated at all. This covers distinct counts and all calculations containing a division operator in their formulas (e.g., average, percentage, ratio). Adding up the results of such a calculation doesn't 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 the margin by the sum of the sales amount (instead of dividing the margin by the sales amount of each individual sales row and then adding up those results).

Formulas can also be applied to nonnumeric values. [Chapter 3](#) discusses 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. To avoid this, you need to convert all flags and indicators delivered by the source system into meaningful text. For example:

- `FinishedGoodFlag` with content 0 or 1 should be transformed accordingly into the text “not salable” or “salable.”
- `Productlines` R, M, T, or S should be transformed accordingly into the text Road, Mountain, Touring, or Standard.
- `Class` column entries 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, avoid blank cells (or null) for texts, and replaced with meaningful text: “unknown”, N/A, “other”, etc. Depending on the context, a blank value could be transformed to different text (to distinguish an “unknown” value from “other”) within the same column.

Do you create reports on data that 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 that didn’t 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 that doesn’t bear any relation to a point in time. Therefore, a date dimension is common in the majority of data models. The date dimension can be handled in a couple of ways.

Create columns for all variants of date information that will be later used in reports. Year, month number, month name, name of the weekday, week number, etc. are common examples. The report tool need not cover this, but the variant should show the pre-calculated columns. Therefore, add a column for every variation needed in the report (e.g., December 2023, 2023-12, Dec, ...). Sometimes a numeric column to reference the date table is used, instead of a column of data type `Date`. This can be derived by a simple formula: $\text{Year} \times 10,000 + \text{Month number} \times 100 + \text{Day number}$.

Create 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 “[Time and Date](#)” on page 203).



There is no year 0000 in the Gregorian calendar; the first year is 0001 (AD = Anno Domini or CE = Common Era. Before year 0001 comes year -0001 (or: 0001 BC = Before Christ or AC = Ante Christum or BCE = Before Common Era). This makes calculations into the past easy (as the same rules apply to AD and AC), but it sometimes confuses people that the second millennium ended in the year 2001 (as it started in year 0001) and not in 2000.

Calendar week numbers can be tricky, by the way. Apart from the fact that about half of the population of this planet start their weeks on Sundays, while the others start on Mondays, there are basically two definitions of how to calculate the calendar week numbers. These definitions deviate from each other only in certain years, which could be the reason that you do not discover a possible mistake. If you don't pay attention to the calculation of the calendar week, you might end up with a sudden surprise in one year. Wikipedia's got you covered in case you need to find out which definition is the one the report users expect ([ISO week date](#)).

A time dimension (having rows for hours and minutes within a day), on the other hand, is in fact 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 1,440 (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 525,600 (365 days times 24 hours times 60 minutes) rows. For every year, you would add another 525,600 rows in the datetime table.

Talk to your end users to find out on which granularity level of time they need to filter and group information. If the finest granularity is, for example, 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 entity can play different roles in a data model. A person could simultaneously be an employee and a customer as well. A year could refer to the date of an order and/or of the order's ship date. These are examples of role-playing dimensions.

You can assign different roles by loading the table only once, and then assigning different roles by creating several relationships between the dimension table and the fact table, according to the roles. For example, if you create two filter relationships 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 a second with the Sales' ship date, then the report creator will need a way to specify which role the dimension should play in a visualization.

You can also assign different roles by loading the table twice into the data model, under two different names. For example, you would load the Date table first as an Order Date and second as a Ship Date table. (Make sure that the column names are unique throughout the data model, by, for example, 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 their needs.

Slowly Changing Dimensions

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

We talk about slowly changing dimensions when attribute changes happen 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 isn't a clear line here on how to distinguish slowly changing dimensions from rapidly 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, Type 1, etc.

Type 0: Retain Original

Usually only a small set of dimensions (and their columns) will never change. For example, August 1, 2023 will always be a Tuesday and will always be part of the month of August and the year 2023. This will never change—it's not necessary to implement a way to update 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 referring to the 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) input 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 the 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 of. 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 happen). All sales before the change use the old version’s surrogate key as the foreign key. All sales after the change use the new version’s surrogate key as the foreign key.

Querying is therefore not that big of an issue (as long as the report users don’t need to select a certain version of the customer to be used for a report for a given point in time; “[Slowly Changing Dimensions](#)” on [page 207](#) covers how to implement this request). In [Table 2-4](#), you see a table that contains the additional columns to describe the timespan when the row is/was valid (`ValidFrom` and `ValidUntil`). This looks somewhat 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 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 created 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 2 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	



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 Attributes

Instead of creating a new row for every change, Type 3 keeps a dedicated column per version. Obviously, you need to decide up front 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’ve 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-Dimensions

This approach keeps track 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 so in the extra table.

Again, I have never implemented this type of slowly changing dimension for one of my customers, but you may still find it useful.

Types 5, 6, and 7

The rest of the types are more or less combinations of the previous versions. I'm sure they have their use cases, but I never had to implement them; Type 1 and Type 2 have been sufficient for my client's needs so far. That's why I just give you a short overview here instead of an 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

You can find more about these types in Margy Ross's article, "[Design Tip #152: Slowly Changing Dimension Types 0, 4, 5, 6 and 7](#)," at the Kimball Group.

Hierarchies

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

- Product categories (and their main and subcategories)
- Geographical information (like continents, regions, countries, districts, etc.)
- Time and date (year, month, day, hour, minute, second, etc.)
- Organization tree (every employee reports to another, up to the CEO)

I'm convinced that you'll have some hierarchical structures in your data model(s) as well. From a technical perspective, the latter example (organization tree) is different from the other examples. Typically, you store year, month, day, etc. in separate columns of a dimension table to represent such a *natural hierarchy*. This doesn't necessarily apply to an organization tree, which is a so-called *parent-child hierarchy*.

There are plenty of ways to store parent-child relationships in a table. One 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 traverse the organization tree from one row to the other.

You can either use a recursive *common table expression* (CTE) written in SQL to collect information from different levels or write a recursive function in T-SQL (I demonstrate both in “Hierarchies” on page 395). You can also solve this in DAX (“Hierarchies” on page 210) and Power Query (“Hierarchies” on page 286). Either way, Power BI asks you to create a so-called materialized path per row in the employees table. Figure 2-1 shows an example of what the materialized path could look like for a bunch of nodes in a hierarchy.

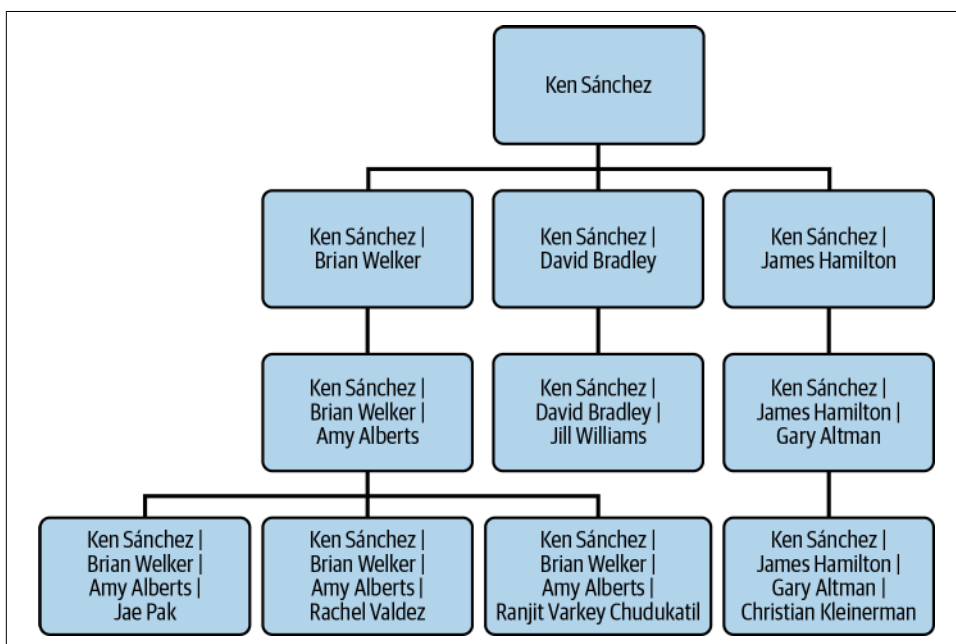


Figure 2-1. A materialized path

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 delimiter. I used the full names for better readability, but in reality you should use the primary keys of the nodes (e.g., EmployeeKey) instead, of course. The delimiter 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 delimiter 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 CEO’s name will then appear in the column representing level one of all employees. Level two will contain the next management level, and so on. You could 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	Level 1	Level 2	Level 3
290	277	Amy Alberts	112 277 290	Ken Sánchez	Brian Welcker	Amy Alberts

You made it through [Chapter 2](#), and now it’s time to wrap it up.

Key Takeaways

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

- Denormalize your fact tables and fully normalize your dimension tables to form a star schema.
- Push transformations upstream as far as possible and delay calculations as long as possible, preferably at runtime if performance allows, and still in a centralized semantic model (not duplicated in every report). The more upstream you put definitions, the more people and tools can use it, which avoids having calculations defined on many places.
- Avoid keeping flags and indicators as they are but transform them into meaningful texts instead.
- Create time and date 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); they play a crucial role in many data models.
- A single dimension may play more than one 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.
- Model slowly changing dimensions 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, Type 2.

- Use two different types of hierarchies. In one, you have one column per level (either within one table in the case of a denormalized table or every column in a different table in the case of a normalized model; this first kind is called a *natural hierarchy*). In the other type, a child row references its parent row, both stored in the same table, called a parent-child hierarchy.

Real-World Examples

Chapters 1 and 2 introduce 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 those steps. Ensure that your data model is built upon these steps before you dive into this chapter's use cases.



If you didn't take all the steps in Chapters 1 and 2, applying the concepts of this chapter might be frustrating because the best basis for advanced concepts is the star schema I describe in those earlier chapters. There is no shortcut to the advanced topics.

In this chapter, I describe five use cases that I encounter when working with many of my customers; I therefore assume that there's a high likelihood you'll face them as well sooner or later. The list is, of course, a personal selection; every new project comes with its own challenges, and I can't predict every unique circumstance you'll run into.

"Binning" on page 58 describes the challenge of not showing the actual value, but a category the value falls into instead. In "Budget" on page 60, I use the case of budget values to introduce a data model containing more than a single fact table (which still conforms with the rules of a star schema). I'm excited (and proud of) "Multi-Language Model" on page 63, which describes the problem of giving the report user full control over the display language of the report (i.e., headlines and content) and my solution. It's similar to "Key-Value Pair Tables" on page 65. A data source for a data model of one of my customers contained all needed information in a single, unpivoted table, and I describe the problems I encountered and (semi-)automatic solutions to pivot the table, which I could not find a predefined method for.

Finally, “Combining Self-Service and Enterprise BI” on page 67 describes the basis for what can be done in Power BI and Analysis Services in a *composite model*. In the first use case, I describe different approaches to 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 don’t want to show the actual values (like a quantity of four) but a category instead (like “medium” or “between one and five”). Basically, we have the options in the following sections to model this requirement, all of them with different disadvantages and advantages.

Adding a Column to a Fact Table

You might add a new column to a table containing the value and make 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 because you would make the fact table (containing the value to be binned) wider. Also, if the range of the bin or its descriptive text changed, you would need to update the fact table. In some implementations, this won’t be possible because of the fact table’s size and the update statement’s runtime.

Table 3-1. Adding a column to a 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

You can create a lookup table like 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 I describe in Chapter 2, when we apply a lookup table to transform a flag or code into meaningful text.

You then create an equi-join between the table containing the values to be binned and this lookup table. This looks a bit unusual; we’re used to joining primary keys and foreign keys and not, e.g., a quantity. But this solution is easy to implement and performs very well.

Table 3-2. Adding a lookup table containing distinct quantities and their bins

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

Another advantage is that such a table is usually easy to create. Maintaining the table is easy, in principle, as well. The only catch is if somebody needs to maintain the table by hand and makes typos (then a value of three might be assigned to “medum” instead of “medium” and would be shown as a unique category for itself). Or, if the categories can get mixed up by accident (say 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 drawback is that this idea only works if we can generate a distinct list of all possible values, though. Yes, you can add some extra lines for outliers (quantities beyond a thousand, maybe), but if we aren’t 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, however, help to overcome this problem.

Describing the Ranges of the Bins

The other option is to create a table containing three columns: one defining the lowest value per category, another one to define the upper value per category, and finally a 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 bins

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 involves some extra logic when assigning the actual values to the category.

I strongly recommend making one range's assigned 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 than or equal to 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; a value is either lower than the upper bound (and therefore falls into this category), or greater than or equal to the upper bound (which matches the lower bound of the next category) and therefore falls into the next category.

Another challenge I often see in models I build for customers is combining information of different granularity in a single data model. This is the case when you combine actual values and their budget, as I discuss next.

Budget

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

The definitive goal of a star schema is to add new information to only the existing tables, if possible, and to avoid creating a new table for every extra piece of information. Joining tables is an expensive operation in terms of report/query runtime. That said, you should first evaluate if the granularity of the new information matches the granularity of an existing table.

Identifying the Granularity

Let's first look at cases, where you can add information without further changes. Maybe you want to add the information about a product's category to the reports (and therefore to the data model). If you already have a table of the same or lower granularity than the product category (e.g., a dimension table `Product` that contains information about individual products), you can simply add a `Product Category` column to that dimension table. The granularity of the `Product` table won't 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 information you want to add to a fact table is of the same granularity, you can simply add it as a new column. For example, in a table containing Sales amounts in EUR, you can add a column containing the Quantity in pieces. As long as both the amount in EUR and the quantity in pieces are of 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

In the next section, we'll look at more challenging cases.

Handling Fact Tables of Different Cardinality

If the table you start with is on the level of 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. It's expected that there are several products (with individual rows) per Product Category Key. The cardinality of relationships from (fact) tables applied to the dimension table would suddenly change from one-to-many to many-to-many: for each row in the fact table, there'd be several rows in the dimension table. This is best avoided (see "Relationships" on page 6). Instead, keep the existing dimension table at its current granularity and introduce a new dimension table with the different granularity.

Table 3-6. Table for product categories

Product category key	Product category
10	Group 1
20	Group 2

Something similar happens if you want to add facts of a coarser granularity. While we collect actual sales information at the level of granularity of day, product, customer, etc., values for a budget are typically available only 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 finer 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 (Table 11-3), 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 is the Product Group the primary key).

Table 3-7. A budget is typically of a different granularity than the actual values

Month	Product Group	Budget
2023-08	Group 2	20,000
2023-08	Group 3	7,000
2023-09	Group 2	25,000
2023-09	Group 3	8,000

No matter the reason for a many-to-many cardinality, it’s best practice to introduce a table in between to bridge the many-to-many cardinality and create two one-to-many relationships. 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 of this table to the Budget table then has a one-to-many relationship. Likewise, the relationship from this table to the Product table is one-to-many as well (Figure 3-1).

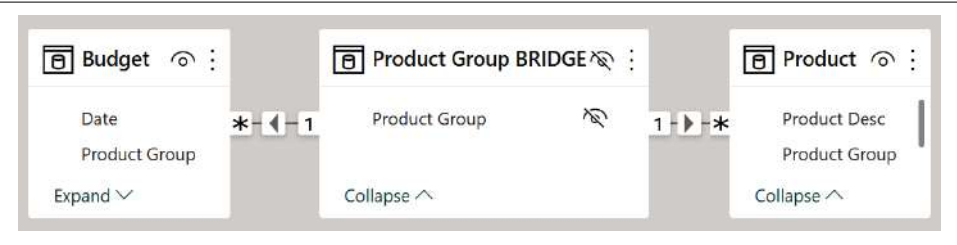


Figure 3-1. Bridge table

In the end, the relationship between tables `Budget` and `Product` still has a cardinality of many-to-many but is split into two one-to-many relationships, which makes handling in many tools easier.

Multi-Language Model

In a global context, your users might expect to get reports in a language of their choice. Let’s look at a data model that allows for such. It will take extra care to localize the content of the report’s data model and software. These are the moving parts of a multi-language solution:

Text-based 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. In our example, we can see that for a `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. 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 headlines)

Because I don’t want to create and maintain several versions of the same report (one for each and every language), I store all the text for visual elements in the database as well (in a table like [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 `Display Text` for a specific text identifier.

Table 3-9. 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 aren't strictly translated, but localizing in a broader sense requires converting between currencies. There are a wide variety of solutions when it comes to finding the correct exchange rate. In a simple model, you'd have one exchange rate per currency (see Table 3-10). In more complex scenarios, you'd have different exchange rates over time and an algorithm to select the correct exchange rate.

Table 3-10. 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 databases allow you to translate the names of all artifacts 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 metadata. And usually, they understand terms in English (or in the language the data model was created in). Report consumers won't directly see any part of the data model but what the report exposes to them. And a report can expose text via translated visual elements. Therefore, in my experience, the use case for metadata translation is only narrowly applicable.

User interface (e.g., Power BI Desktop)

You need to check the user documentation on how to change the UI's language. In **"Multi-Language Model"** on page 139 I will describe the settings for Power BI Desktop, the Power BI service and Power BI Report Server.

Some data sources expose their information in a way that looks like a table on first sight, but which—after a closer look—turns out not to be a classical table with information spread out over columns. You'll learn how to handle such tables next.

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 (hence 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` entries 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 you what kind of content to expect in the `Value` column.

Table 3-11. 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 a `Key` and its `Value`). There's 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).

However, there’s one 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 of 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”). This will show one Value on the rows and the other on the columns of the pivot table (or a *matrix* visual in Power BI, for that matter) as well as 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 with 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 correlations of combinations of any attribute. You will see this in action in “Key-Value Pair Tables” on page 149.

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’re 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 wherein 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	firstPurchase
1	Bill	Seattle	20,000	1980-01-01
2	Jeff	Seattle	19,000	2000-01-01
3	Markus	Alkoven	5	2021-01-01

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 laying out the 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 a 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 or uses no-code/low-code services in the cloud).

Everything available in an *enterprise BI* solution, on the other hand, is built with heavy involvement of an IT department. Servers are set up. Services are deployed or configured. 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 running the necessary services). But this takes time to build. Sometimes, collecting all the requirements and writing down the user stories for the engineers to implement will 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 that hasn't made it into the data warehouse (yet). That's where self-service BI comes in.

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. “**Key-Value Pair Tables**” on page 149 describes 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 concepts for how to solve them:

- Bin values with either a simple lookup table (which contains all possible values and their bins) and physical relationships between the values and the lookup table, or describe the ranges per bin and apply a non-equi-join between the values and the lookup table.

- Add new tables to a data model only if the information can't be added to an existing table. As a budget is usually on a different granularity level than the actual data, I cover this as a use case for a multi-fact data model.
- Cover content of textual columns, text on the report, currency exchange rates, the names in the data model, and the language of the UI of the application you're working with when you want to implement localized reports.
- Combine self-service and enterprise BI carefully. They will always exist side-by-side, and this chapter covered some of the challenges that presents. In [Chapter 8](#), you will see how both worlds can live together in Power BI.

You'll learn how to implement the solutions in DAX, Power Query, and SQL later. First, I introduce ideas and concepts that allow you to optimize a data model for performance in [Chapter 4](#).

Performance Tuning

You are very blessed if performance tuning has never been a topic in a report you built. Usually it's not a question of *if*, but *when* the performance of a business solution becomes a priority.

Generally, if you've taken the previous chapters seriously and transformed all data into a star schema data model, you've made an important step toward well-performing reports. The shape of the data model plays an important role when it comes to performant reports. But, of course, many more pieces help determine how quickly reports return data or react to filters. Because this book is about data modeling, I limit myself to discussing performance-tuning topics only as they relate data modeling.

My first computer had a Turbo button on the front of its case, next to the power button. I 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. Why build a model that doesn't run as quickly as possible? You should always keep performance in mind when building your data models.

Unfortunately, there's no Turbo button in Power BI to hit after powering on. But there are concepts you can apply.

If you're about my age, you may have had a paper list of phone numbers for your family, friends, and neighbors in your youth. Mine had the most important people first, and I added more and more people to it later. When the list got to a decent length, scanning it every time I needed a number frustrated me. So I started a new list and split the names and numbers onto different pages: one page per letter in the alphabet in alphabetical order by first name.

This principle applies to databases as well. You can create simple tables, where new data is (chronologically) added at the end of the table (or in between rows after a row

is deleted). Adding data to the table takes very little time because there's no need to find the right place; you can just use the next empty space. But you pay a penalty when you read from the table because its full content has to be scanned for every query. Filters will only reduce the result set, not the process.

The alternative is to store all the rows in a table in a certain order. As long as your filter refers to the order key, finding the right rows can be faster: you ignore all the rows that don't 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; databases store metadata for the sake of speeding up queries. But writing data into such a table will be a bit slower: the right position for the new rows has to be found. New space may need to be made available at this position. Metadata must be maintained.

These examples should make a very important principle clear: you can exchange query speed for space on disk or in memory. Speeding up queries this way will likely slow down write operations. In analytics, reading data is most often done frequently and quickly, while refreshes (write operations) can be done only at scheduled points in time and are fewer in number. 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 for storing data in tables:

Storing only queries

You could opt to not physically store (duplicate) data, but keep the data in the original tables. Instead of the data, you store only the query that 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 have 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 because 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 metadata

You can distinguish between metadata automatically added by the database system (like descriptive statistics) and metadata explicitly added (like indexes). A database index is like the index at the end of this book. Instead of scanning the whole book for the term "foreign key," you can jump to the index, where important terms are ordered alphabetically. In the index, you can quickly discover whether the book covers this term and find page references where you can find

more information about it. 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 than querying by a sort order that doesn't match the table's, or without a covering index. 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 were already stored (persisted) 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 employ a strategy to exchange disk space for query runtime, and therefore increase the duration of time to process the transformation and refresh the user-facing data model.

Key Takeaways

A good data model takes query performance into account. By following the principles of the earlier chapters, you've already created a data model with good query performance. No matter which data model you design or which tools you use, you have a wide variety of possibilities to control the performance by applying a taste of three 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. Transformations 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.

In **Part II**, we leave the world of concepts and dive into Power BI Desktop and its possibilities for creating the data model, which will make the lives of your report creators easier.

Data Modeling in Power BI

Chapters	Parts				
	Part 1 Data modeling in general	Part 2 Power BI	Part 3 DAX	Part 4 Power Query	Part 5 SQL
Understanding a data model	Chapter 1	Chapter 5	Chapter 9	Chapter 13	Chapter 17
Building a data model	Chapter 2	Chapter 6	Chapter 10	Chapter 14	Chapter 18
Real-world examples	Chapter 3	Chapter 7	Chapter 11	Chapter 15	Chapter 19
Performance tuning	Chapter 4	Chapter 8	Chapter 12	Chapter 16	Chapter 20

This second part of the book concentrates on the specialties of Power BI's features when it comes to data modeling. I will walk you through the basic concepts, problems, and solutions. **Part II** keeps DAX, Power Query, and SQL out of the game as much as possible—their specialties are discussed in later parts of this book.

First, I will introduce you to the *Model view* and its parts in **Chapter 5**:

- Tables
- Columns
- Relationships

You will learn that Power BI doesn't use the terms *primary key* and *foreign key*, but they still play an important role when it comes to relationships and their *cardinality*.

In the Model view, you won't combine tables, only define their relationships. Still, it's important to understand how to build a data model, which is easy to understand and allows for performant queries later.

Chapter 6 is all about building a data model, which works optimally in Power BI:

- Finding the right way to normalize and denormalize tables
- Telling Power BI the formulas of your calculations
- Providing Power BI with a date (and optionally a time) table
- Implementing a solution for tables, which play more than one role inside the data model
- Taking care of *slowly changing dimensions*
- Combining columns into a hierarchy

I pick up the real-world use cases discussed in **Chapter 3** and go over some important concepts that you'll use in Power BI's Model view (**Chapter 7**):

- Binning values
- Multi-fact data model (a data model that contains more than one fact table)
- Multi-language data models
- Key-value pair tables

The last chapter in this part (**Chapter 8**) will talk about the Model view's options to achieve good query performance with your data model: basically, you can decide if you want to store a copy of all the data in Power BI and refresh it regularly, or query the data source every time a visual is shown. You will learn about the advantages and disadvantages of these *storage modes*.

Understanding a Power BI Data Model

In this chapter, you will learn how to create a useful data model in Power BI (and the Analysis Services tabular model). This chapter concentrates on the features of the Model view. The following parts of this book discuss options for bringing data of any shape into the desired shape in Power BI (see [Chapter 1](#) for a general description). You will learn that Power BI needs a data model to work.

I will detail the properties tables can have and how to put them into relationships with each other. You'll find out there's no need to explicitly mark primary and foreign keys, but you still must be able to identify them to create appropriate relationships. The cardinality of the relationships plays an important role in Power BI. Luckily enough, you don't need to think about the joins and join path problems too much. You only need to create relationships for your data model. Power BI will automatically use these relationships to join the tables appropriately when the data is queried for a visual. Power BI will also make sure to execute requests against the data in a way that the join path problems do not occur (see [“Join Path Problems” on page 20](#)).

In this chapter, I reiterate 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 that 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, select the Model view in Power BI (or the *diagram view* in Visual Studio in the case of the Analysis Services tabular model). This view looks much like an entity relationship diagram (ERD) but has subtle differences we discuss in [“Entity Relationship Diagrams” on page 100](#).

The “All tables” tab shows each and every table in the data model, as shown in [Figure 5-1](#). For bigger data models (those 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 wise 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, all tables might not fit on your screen (or only if you zoom out so much that you can’t read their names). A separated layout with less content can be helpful.

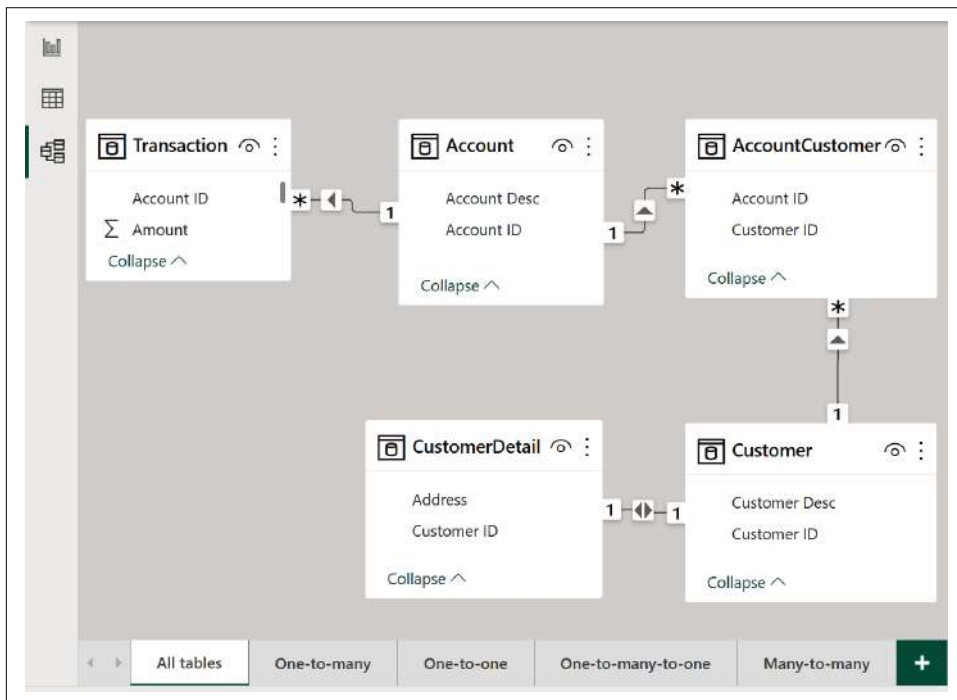


Figure 5-1. Model view

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

The Model view has three properties (see [Figure 5-2](#)):

Show the database in the header when applicable

This is applicable in data models in *DirectQuery* and *Dual* modes. You can learn about the different storage modes in [Chapter 8](#). This setting is turned off by default.

Expand or collapse tables

Each table can either be expanded or collapsed. When a table is collapsed, no columns are shown unless you select “Show related fields when card is collapsed.” Related fields are columns, which are used in relationships. This property is enabled by default.

Pin related fields to the top of the card

As you might guess, pinning related fields to top of the card will result in columns that are part of a relationship being shown on the 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).

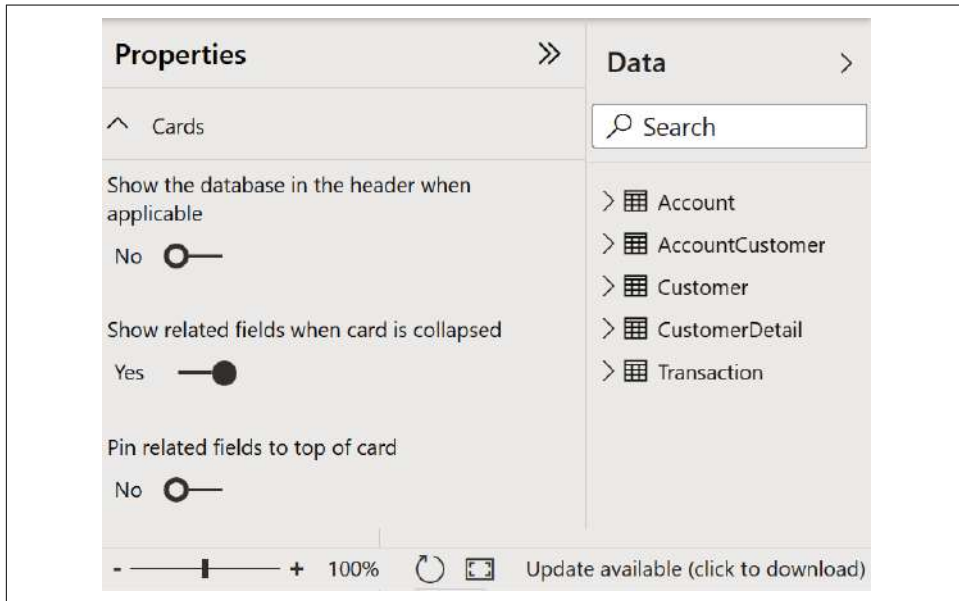


Figure 5-2. Properties in Model view

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

Basic Concepts

This section introduces the basic concepts of a data model:

- Tables and columns
- Relationships
- Primary keys
- Surrogate keys
- Foreign keys
- Cardinality

Tables and Columns

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 table name, the columns (fields) are listed in alphabetical order.

A table offers a list of functionalities, which you can access either by right-clicking the table name or left-clicking the ellipses (...), as in [Figure 5-3](#):

Add related tables

This option will add all tables that have a filter relationship with the chosen table to the Model view. This option is only available when the table has relationships to other columns, and only in a layout view (not in "All tables").

Select New measure or New column

You can create these within the selected table. In ["Calculations" on page 190](#), you can learn more about these two options.

Refresh data

You can refresh the content of the whole table. The data source has to be available at this point in time.

Edit query

"Edit query" will open the Power Query window. In [Chapter 13](#), I introduce the capabilities of Power Query.

Manage relationships

In ["Relationships" on page 88](#), you will learn everything about the options available via the "Manage relationships" dialog.

Incremental refresh and Manage aggregations

See [Chapter 8](#) for details on incremental refresh and "Manage aggregations."

Select columns

You can select all columns in this table and then change the properties for all of them in one go.

Select measures

This will select all measures in this table. You can then change the properties for all of them in one go.

Delete from model

If you choose this, the whole table will be removed not only from the layout view, but from the whole file (incl. Power Query).



Be careful because “Delete from model” cannot be undone. Make sure to save intermediate versions of your changes to have a backup.

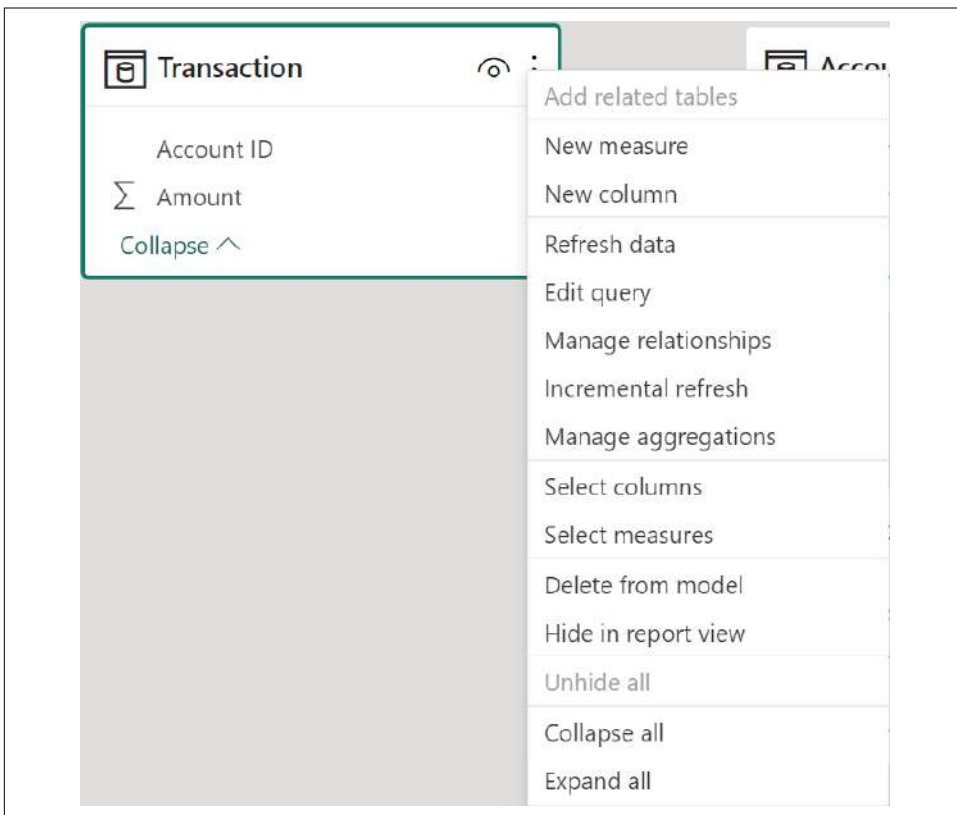


Figure 5-3. Model view context menu

Hide in report view

With this option, the table (and all its columns) are hidden. The goal is to avoid overwhelming the report creators with tables (or columns) holding intermediate results not intended to be shown in a report.



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 that are only helping to build the data model and don’t contain any content that should be used in a report (e.g., to show the value, filter on it, etc.).

Remove table from diagram

This option is only available in layouts not in tab “All tables.” This does not remove the table from the whole file but just from the current layout.

Unhide all

With this option 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

This 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

This option 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 measures and hierarchies as well. Measures are written in the DAX language. Chapters 9 through 12 show many capabilities of the DAX language, and you can find more information about measures in “Calculations” on page 190. Hierarchies group several columns into one entity. “Hierarchies” on page 129 in Chapter 6 is dedicated to hierarchies.

Tables have properties in Power BI, as you can see in Figure 5-4:

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 “Tables or Queries” on page 244).

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 5-4. Model view table properties

Description

The description is shown in a tooltip when you (or the report creator) 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. You should add alternate terms your organization uses (e.g., if *revenue* is an alternative for *sales*, enter “revenue” as a synonym for “sales”).

The list of synonyms helps the Q&A visual to answer your questions. Visuals are out of scope for this book, but you can find an in-depth description of the Q&A visual in my book *Self-Service AI with Power BI Desktop* (Apress, 2020).

Row label

This field helps in the Q&A visual when you reference a table. It will then show the content of the column you specify here. Select the column containing the name of the entity as the row label of the table. It helps in Excel in a similar way too. (See “[Set Featured Tables in Power BI Desktop to Appear in Excel](#)”.)

Key column

Again, this feature helps in Excel. Select the column containing the primary key as the key column of a table. You can only select a column that doesn't contain any duplicate values.

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 that do not contain content relevant for the reports, but that are needed to create the data model in the right shape, e.g., *bridge* tables.

Technically, tables can't be hidden in Power BI or the Analysis Services tabular model; only columns can. If a table contains only hidden columns, then the table is hidden as well. Hiding a table changes the “Is hidden” property of all columns within the table. If you unhide a table, all columns will be visible no matter whether they were hidden before you changed the setting on the table level.



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

Is featured table

This makes the table easier to find in Excel. It'll show up in the “**Excel Data Types Gallery**”. Tables where this setting is disabled can still be found via *Analyze in Excel* (see **Microsoft's documentation**).

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 5-5**. In the Model view, you set the following 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 “**Tables or Queries**” on page 244.

Description

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

Synonyms

The Q&A visual uses the provided synonyms to find columns not directly referenced by their (technical) names, but by alternative names and versions as well (e.g., if a user might query Q&A for *revenue* and the column is called `Sales Amount`, add *revenue* to the list of synonyms for the column `Sales Amount`).

Display folder

In tables with a lot of elements (measures, columns, hierarchies, etc.), browsing 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`, just enter “`Sales`” into the “Display folder” field. You can even create subfolders by using the backslash (\) in this field. For example, “`KPI\Sales`” puts the column into a folders `KPIs` and a subfolders `Sales`.

Is hidden

This hides the column. You should hide all columns needed for the data model only (e.g., all keys), but that should/will never be shown in a report.



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

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 5-5. Model view column properties

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 the format in which the value of the column is shown. The options are dependent on the data type, of course: “Percentage format,” “Thousands separator,” “Decimal places,” “Currency format,” etc.



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 one only showing the date. The time portion is still stored in the data model (which is expensive in terms of storage; see [“Tables” on page 6](#), and might break filters if the dimension table doesn’t contain the time portion, but the fact table does). If nobody will ever report on the time portion, it’s 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 impractical. Typically, you don’t want a list of months sorted alphabetically. To use this property, 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; it 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. For example, if you mark a column as a `Place`, Power BI will suggest showing the content on a map visual. It also gives Power BI information about what the content means (e.g., if a two-letter code is identifying a US state or an ISO country code).

Summarize by (Advanced)

While this setting appears in the Advanced section of the UI, I consider it ideal for newcomers to Power BI’s data modeling capabilities. It allows you to specify a default aggregation function that should be applied when you add a numerical column into a visual. For example, if you add the `SalesAmount` column to a visual, you usually don’t want to get a (long) list of rows of the `Sales` table showing individual `SalesAmount` values. You want a total. You can override the data model’s default aggregation setting per visual (e.g., showing the average of the `SalesAmount`). “Summarize by” allows you to specify how the values are aggregated: `Sum`, `Average`, `Min`, `Max`, `Count`, and `Distinct Count`. For numeric values that you don’t want to aggregate (e.g., a year or date), you must specify `None`.

Any setting besides `None` will create a so-called *implicit measure* containing the chosen aggregate function. Unfortunately, implicit measures do not work with all

client tools (they have worked in Excel since 2023). Implicit measures are automatically disabled as soon as you create calculation groups (you can learn about them in [Chapter 10](#) in the section about calculations in DAX), breaking existing reports, which depend on those implicit measures.

My strong recommendation, therefore, is to explicitly create measures for columns for which you need to apply aggregations and set “Is hidden” to Yes for such columns. (You can learn more about explicit measures in [“Calculations” on page 190](#) as well.)

Is nullable (Advanced)

This specifies if the column may contain blank cells (or null cells, as termed in relational databases). If you consider blanks in this column to be a data quality issue, then you should turn this setting to No. Doing so will result in errors during refreshes whenever a row contains blank cells for this column. Every row of a column must conform to the column’s data type in Power BI.

Let’s take a closer look at the different data types Power BI allows you to choose from:

Binary

This data type is not supported and exists only for legacy reasons. You should remove columns of this data type before loading data into Power BI and Analysis Services or just delete instances 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 can also contain *blank* values, which represent an unknown value.



For databases, it’s typical that every data type also supports an “unknown” value. In relational databases and in Power Query, this is represented by *null*, and 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 (January 1, 1900).

That something is unknown might be important (and should not be set as equal to some default value). In a user-friendly data model, an unknown value should be replaced by something that explicitly tells users that the value is unknown (e.g., string “N/A” or “Not available”), as described in [“Tables” on page 6](#).

Fixed decimal number

This data type 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 with the exact same internal representation with the only difference being that the fixed decimal number is automatically divided by 1,000 before it is shown. Due to the limit to four decimal places, you can face rounding errors when values are aggregated.

Decimal number

This is a 64-bit floating point number that can handle very small and very big numbers, both in the positive and negative spectrum. Because it is only precise to up to 15 digits, you may face rounding errors when values are aggregated.

Date/Time

This represents a precise point in time (to 3.33 milliseconds). Internally in a database, all date- and time-related data is stored as a decimal number counting the days since a specific point in time. (In the case of Power BI, this point in time is midnight of December 30, 1899). The decimal portion represents the parts of the day (e.g., 0.5 represents 12 P.M.). I point 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 P.M.” or “2023-08-01 13:15:00”).

The “format properties” task is used to put a value into a user-friendly format humans can read, but it does not change the internal representation (which is 45,139.55 in Power BI for the given example—and would obviously not be very user-friendly in a report).

Date

This 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 represents August 1, 2023).

Time

This 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 represents 1:15 P.M.).

Text

This holds Unicode character strings. Columns of this type can hold up to 268,435,456 characters.

Whole number

Values of this data type are stored as a 64-bit integer value. This data type doesn't 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.

In many data models, tables don't live by themselves, but contain information that is somehow related to information in other tables. Let's talk more about these kinds of relationships.

Relationships

In Power BI, relationships connect tables with each other and *look like* foreign key constraints in a relational database, but they *work differently*. While foreign key constraints limit possibilities (they prohibit values in foreign key columns that can't be found in the related primary key columns), relationships in Power BI exist solely to propagate filters from one table to another. If you filter the Date table to a certain year, the filter relationship will propagate the filter to the Sales table, so queries will 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 correctly setting the relationships.

Creating filter relationships is rather easy. Power BI offers three methods (automatic creation, drag and drop, and a dialog box), 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 (see [Figure 5-6](#)).

You can let Power BI import 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. It does this by applying a set of rules: the column names must be the same, the data types of these columns must be the same, and the column's value must be unique in at least one of the two tables.



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

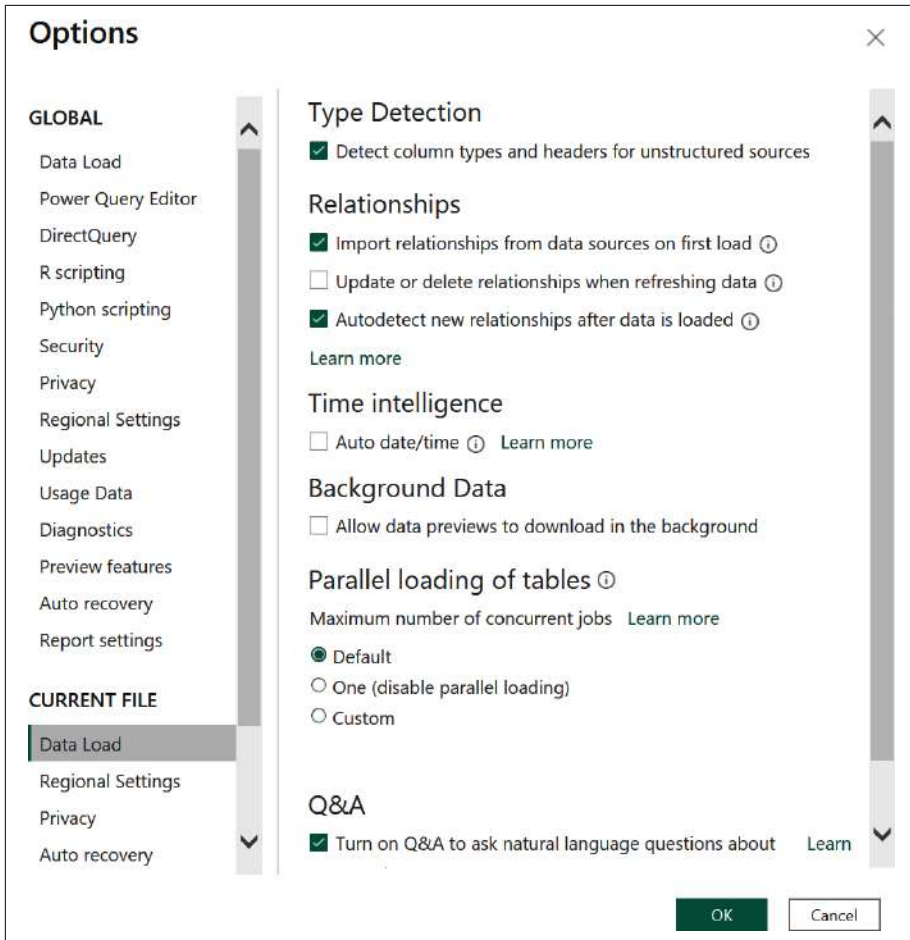


Figure 5-6. Options → Current File → Data Load

Drag and drop

A very simple way to create a relationship is to drag one column over another (from a different table) in the Model view. In theory, it doesn't matter which column you drag over. The cardinality and filter direction are automatically set for you.

It is, however, always a good idea to double-check whether all properties of the created relationship are as they should be. More than once, I've seen Power BI create a many-to-many relationship (due to unintended duplicates in a table) or a one-to-one relationship (due to incomplete 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 to trigger Automatic creation, or Edit or Delete an existing relationship (see [Figure 5-7](#)). By clicking the checkbox Active, you can (de-)activate a relationship.

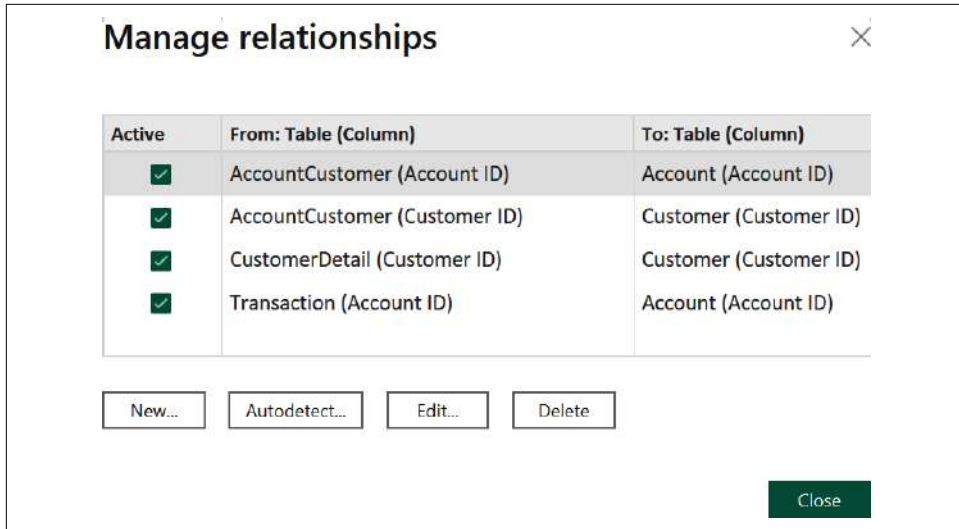


Figure 5-7. Modeling → “Manage relationships”



The order the relationships are shown in 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.

[Figure 5-8](#) shows you the properties 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

Transaction

Column

Account ID

Cardinality

Many to one (*:1)

Table

Account

Column

Account ID

Make this relationship active

Yes

Cross filter direction

Single

Apply changes

Open relationship editor

Figure 5-8. Model view relationship properties



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 ensure that no relationship is defined in a wrong way (read: check the cardinality), that no relationship is missing, and that no unnecessary relationship is created.

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

First table

This is sometimes described as the left table. It's one of the two tables for which you create a relationship. Which table is the first/left table isn't 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 list box. The “Edit relationship” dialog window shows a preview of the values of the first three rows. Selecting more than one column is not possible; Power BI doesn’t allow the use of composite keys.

When you must work with composite keys, simply concatenate the values of those columns (as a DAX calculated column, in Power Query, SQL, or the data source) before creating the relationship. I strongly recommend adding a separator character in between the column values to avoid *false twins*. For example, if you concatenate “12” and “34” for one row, and “1” and “234” for a different row, the resulting concatenated string would be “1234” for both examples. If you put a separator character in between, you’ll get “12|34” in one case and “1|234” in the other.

Second table

This is 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 a given table. “**Cardinality**” on [page 10](#) reviews 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-direction 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 and/or poor report performance.



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

A maximum of one 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'll learn to make use of inactive relationships with the help of DAX. In [“Role-Playing Dimensions” on page 48](#), I show 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 bi-directional filters (which is not recommended; see “Cross-filter direction” in the previous list). Propagation of RLS 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](#).

Assume referential integrity

This property is only available when using DirectQuery (DirectQuery is covered 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 a 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 [“Joins” on page 13](#).

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 [“Binning” on page 216](#), I show ways of implementing non-equi-joins with the help of DAX.

Many, but not all, relationships are built on a primary key in one table and a foreign key in the other table. Let's start looking into how Power BI handles primary keys in the next section.

Primary Keys

In Power BI, you don't 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 not allowed. I strongly encourage you to make sure during the ETL process to have no blank values anywhere, neither in key columns nor other columns, but to replace them with a meaningful value or description (like "not available"). In the "Flags and Indicators" sections of Chapters 10, 14, and 18, you'll learn different ways to achieve this.

Power BI's data model doesn't allow composite keys. If you decide to use a composite key, you need to concatenate all the columns participating in the key into just one column (usually of type *Text*). Make sure to add a separator character between the values. Look for a (special) character that won't ever be part of the column's values (e.g., a pipe symbol, |, when concatenating names). This ensures that the result of concatenating "ABC" and "XYZ" will be 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'd end up with the identical primary key "ABCXYZ" for both rows, which is problematic because Power BI can't then distinguish those two rows from each other.

Surrogate Keys

In Power BI, a relationship can only be created on two single columns in two separate tables; Power BI does not allow the use of composite keys. I strongly advise using columns of type *Whole number* for the relationships because they can be stored more efficiently (compared to the other data types, e.g., *Text*) 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* type. This makes them perfect keys for Power BI. Learn more about creating surrogate keys in ["Surrogate Keys" on page 339](#).

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.

Foreign Keys

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

If you decide to use a composite key as a primary key, you need to concatenate all the columns participating in the foreign key as well. Make sure to concatenate the columns in the very same order as you did for the primary key and 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 available for a single foreign key, and the other way around. This is called *cardinality*.

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. A corresponding row in another table is allowed to be unavailable; for Power BI, it's OK when there is no row in the Sales table for a specific Customer. This is also OK in the real world; 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: it would mean that 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 might be 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 CustomerIDs for the other rows are just valid by chance, but contain the wrong information?



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 on many different projects, one project has many employees), but in all cases where neither of the two columns used for creating the relationship only contain unique values. In case of data quality issues (e.g., duplicated customer rows or multiple rows with a blank CustomerID), Power BI won't let you change the relationship to a one-to-many.

Relationships of many-to-many cardinality are called *weak* or *limited* relationships because they come with two special effects, which are reasons why you should avoid

this type of relationship (use a bridge table instead, as discussed in “Types of Tables” on page 101). The two effects are as follows:

- When there are 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 aren’t shown at all. Reports and the totals might show incomplete numbers. This effect only hits you in case of data quality issues. Avoiding missing rows in the dimension table and invalid foreign keys in the fact table is a good idea anyway.
- Calculations in DAX that use the function ALL (or REMOVEFILTERS) to remove filters won’t remove filters on tables connected over a limited 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 the function ALL in their expression), this problem can appear anytime and can be avoided only by avoiding many-to-many relationships.

I try to avoid **many-to-many relationships**. For very large tables, many-to-many relationships might have better performance than the solution with a bridge table, though.

In the Model view, you can easily spot such problematic relationships: they’re represented with parenthesis-like marks after the cardinality indicators (as you can see in **Figure 5-9**). To avoid all these effects, model a many-to-many relationship via a bridge table. You can learn this technique in “Types of Tables” on page 101.

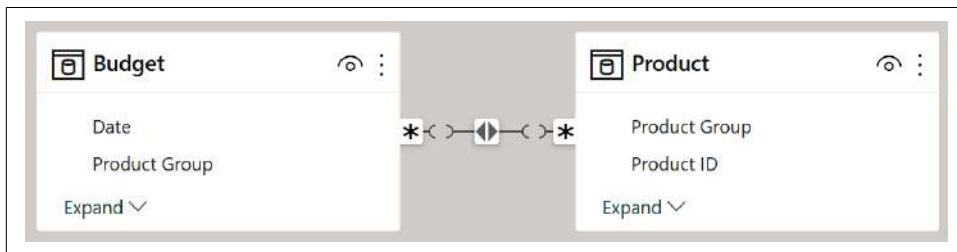


Figure 5-9. Many-to-many relationships are limited relationships, drawn with gaps at both ends



Always ensure that you understand the relationship between two entities in the real world. When you can’t set the configuration for a relationship in Power BI accordingly, double-check why. Don’t just carry on; clarify the business’s requirement and the data quality.

Combining Tables

A data model usually contains more than one table. This section discusses how you can recombine the information of several tables.

Set Operators

Set operators are only available in the Power BI data model when querying data. Jump to the other parts of this book to learn if and how to implement set operators in DAX (“Set Operators” on page 180), Power Query/M (“Set Operators” on page 251), and SQL (“Set Operators” on page 341).

Joins, however, 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. Power BI’s storage engine automatically implements the filter relationships stored in the data model to perform necessary joins. That means if you create reports or write queries in DAX, there’s no need to (explicitly) specify a join operator; the storage engine does this implicitly for you (and the report user).

The filter relationship defines the join predicate (in which two columns are involved). The predicate is always done as an equi-join (the values of the two columns must match). You can’t define non-equi-joins in Power BI’s data model. In [Chapter 9](#), you’ll 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 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 can’t create self-joins at all. In the sections about hierarchies in [Chapters 10, 14, and 18](#), you’ll learn how to flatten parent-child hierarchies so you’re 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 the referential integrity of the model isn’t guaranteed. In relational databases, outer joins come with a performance penalty (compared to inner joins). The storage engine behind Power BI was built with outer joins in mind, so there’s no performance penalty to be expected. There is also no way of comparing—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), I recommend you first guarantee that referential integrity is in place in the data source and then 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 thus make 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 can see in the next section.

Join Path Problems

No worries, Power BI's got you covered on all join path problems: none of the three problems discussed in “[Join Path Problems](#)” on [page 20](#) (loop, chasm trap, and fan trap) are an issue in Power BI:

Loop

You can't create a *loop* directly or indirectly (via intermediate tables); Power BI won't allow you to create multiple active paths. It forces you to declare such a relationship as inactive.

Chasm trap

Power BI has implemented logic to avoid the negative effects of a chasm trap. [Figure 5-10](#) shows a report with three table visuals.¹ The table on top left shows the reseller sales per day. Internet sales per day are on the lower left. On the right, the results of both tables are combined per day. The two tables each have a one-to-many relationship to the Date table (and therefore a many-to-many relationship between themselves). The DateKey column is always taken from the Date table. As you can see, none of the sales amounts for a day (or for the total) are wrongly duplicated, but match the numbers shown for the individual results.

¹ The examples in this section use the [Relationship.pbix](#) file from the book's GitHub repository.

FactResellerSales		DateKey	Reseller SalesAmount	Internet SalesAmount
DateKey	SalesAmount			
20101201	\$489,329	20101201	\$489,329	
20110101	\$1,538,408	20101229		\$14,477
20110301	\$2,010,618	20101230		\$13,932
20110501	\$4,027,080	20101231		\$15,012
20110701	\$713,117	20110101	\$1,538,408	\$7,157
20110801	\$3,356,069	20110102		\$15,012
20110901	\$882,900	20110103		\$14,313
Total	\$80,450,597	20110104		\$7,856
FactInternetSales		20110105		\$7,856
DateKey	SalesAmount	20110106		\$20,910
20101229	\$14,477	20110107		\$10,557
20101230	\$13,932	20110108		\$14,313
20101231	\$15,012	20110109		\$14,135
20110101	\$7,157	20110110		\$7,157
20110102	\$15,012	20110111		\$25,048
20110103	\$14,313	20110112		\$11,231
20110104	\$7,856	20110113		\$14,313
Total	\$29,358,677	20110114		\$14,135
		Total	\$80,450,597	\$29,358,677

Figure 5-10. Chasm trap isn't a problem in Power BI

Fan trap

Power BI has a fail-safe against the fan trap. In [Figure 5-11](#), 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 on the right, you see that the Freight isn't wrongly duplicated per day, but shows the same values as in the SalesOrder Header visual.

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

Figure 5-11. Fan trap isn't 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'll learn next.

Entity Relationship Diagrams

The Model view in Power BI (and the diagram view for Analysis Services projects in Visual Studio) is exactly what you would draw in an ERD if you wanted 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 relationship.

Because the Model view isn't about foreign keys but filters, it also shows the direction of a filter, which can go either in one direction or both directions, represented by a small triangle (for single-directed filters) and two small triangles (for bi-directional filters).

Figure 5-12 shows a single-directed, many-to-one relationship between Account Customer and Customer and a bi-directional, one-to-one relationship between Customer and CustomerDetail, shown in Power BI's Model view.

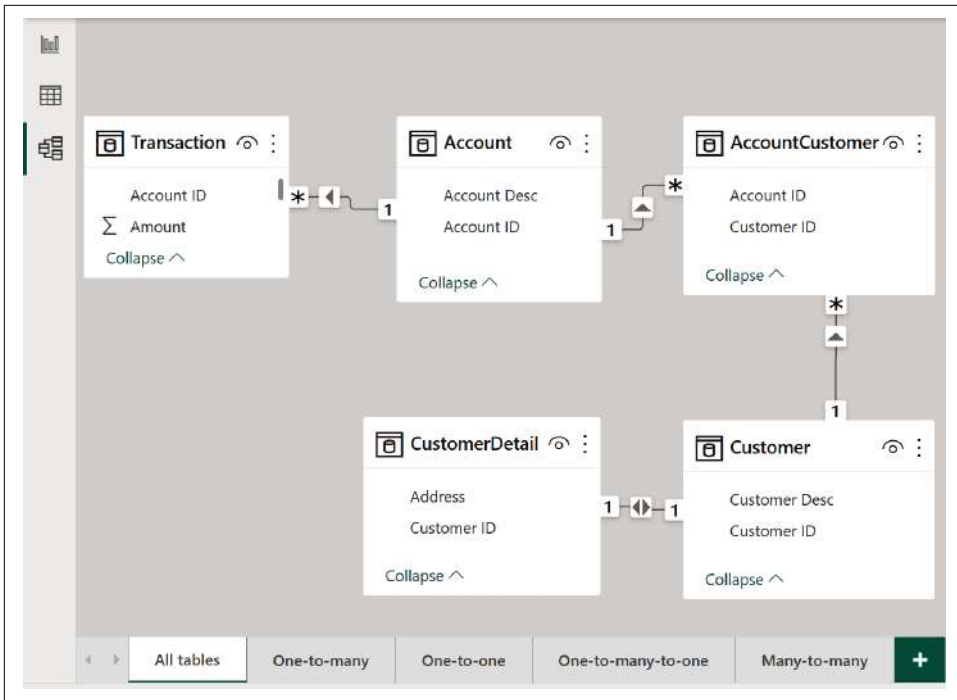


Figure 5-12. Power BI's Model view

Data Modeling Options

As usual, you have a bunch of options when it comes to data modeling. The most important part is to find out what kind of data you want to store to determine the type of table. I share many reasons not to put all the data into a single table when creating a data model for Power BI. You'll also learn how to apply the concepts of normal forms, dimensional modeling, granularity, and ETL to Power BI as well.

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 strongly believe that the type of a (unhidden) table should not be indicated by hints in its name (e.g., Fact_Sales or Dim_Customer). I recommend that table names be user-friendly (report creators usually don't care about what role a table plays in the data model, as long as it's returning the expected results).

The role of a table is simply 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 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 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.

Recall the many-to-many relationship between tables Budget and Product in [Figure 5-9](#). This relationship has a many-to-many cardinality because the budget wasn’t created on the Product table’s level of granularity, but per Product Group instead. The Budget table’s foreign key Product Group is not referencing the Product table’s primary key (Product Key). In the Product table, the Product Group column isn’t 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 many-to-many cardinality. See [Tables 5-1](#) and [5-2](#).

Table 5-1. 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

Table 5-2. A budget is typically of a different granularity than the actual values

Month	Product Group	Budget
2023-08	Group 2	20,000
2023-08	Group 3	7,000
2023-09	Group 2	25,000
2023-09	Group 3	8,000

Relationships have some disadvantages in Power BI, as described in [“Cardinality” on page 95](#). A bridge table resolves a many-to-many relationship and is put between two tables that are logically connected by a many-to-many relationship. The bridge table replaces a many-to-many relationship with two one-to-many relationships. It’s always on the one-side of the two relationships. The bridge table contains a distinct list of key(s) used to join the two original tables.

Because the content is relevant only for creating the relationship, but not for building a report, the bridge table should be hidden from the user. I usually put the suffix “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 (see [Table 5-3](#)).

Table 5-3. Bridge table for product categories

Product Category
Group 1
Group 2
Group 3

Figure 5-13 shows the Model view of three tables of different types. The table on the far left is a fact table (Budget) on the “many” side of the relationship. To the right of that, you see a bridge table (Product Group BRIDGE), which bridges the many-to-many relationship between the Budget and Product tables. The bridge table is on the one-side of both relationships. The table on the far right is a dimension table (Product). It’s on the “many” side of the relationship—the Budget table isn’t referencing the Product table’s primary key (Product Desc), but the non-unique column Product Group.

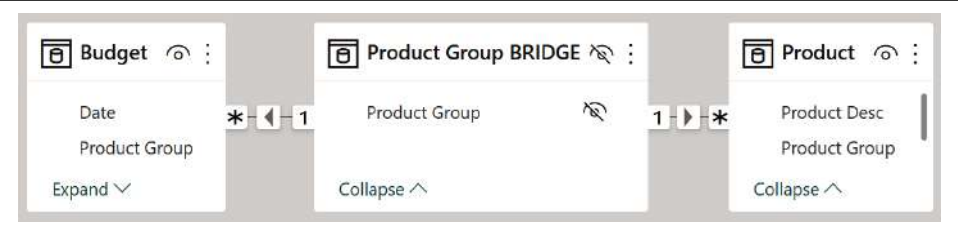


Figure 5-13. Tables of different types

You’ll learn more about the specifics of this data model in “Budget” on page 135. Here, I just used it to demonstrate different table types.

Why should you bother with different (kinds of) tables when you can just just store everything in a single table? The next section explains why that’s a bad idea when it comes to Power BI and the Analysis Services tabular model.

A Single Table to Store It All

While putting all information into a single table is common and allowed even in Power BI, I 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 to split the information in at least two separate tables:

Size of model

Because Power BI’s storage engine stores imported data in memory, Microsoft makes 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 well.

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 store only 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 it will put on your CPU.

Transforming a single table into a star schema will help you fully benefit from the storage engine. The model size will be smaller; the reports will be faster.

Adding new information may be difficult

To extend the data model with additional information, you would need to implement a transformation to add the data to the existing table—which can be dreadful (you need to align the different granularities of the existing and the new information)—increasing the problems you face with the single table. Or, you could add the new information as a separated table. This will only work if you join the two tables on one single dimensional column because you can't have more than one active relationship.

Joining two fact tables directly isn't recommended due to the size of the tables. Transforming the single table into a star schema will make extending the model easier. You would just add new dimension tables, re-use existing dimension tables, and connect new fact tables to the existing dimensions.

Wrong calculations

All clients want correctly reported numbers. Due to some optimization in the storage engine, queries on a single table might result in incorrect results, as [Table 5-4](#) shows.²

Table 5-4. 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

Next, we'll look at 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 column. One version

² The examples in this section use the [Auto-Exist.pbix](#) file from the book's GitHub repository.

removes all filters from the 'Date'[Date] column and the other removes the filter from the 'Sales'[Date] column (by applying function REMOVEFILTERS()).

Figure 5-14 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 sale for the filtered date of February 2, 2023.



Figure 5-14. A report showing the count of rows of the Sales table

Figure 5-15 shows the formula and the content of [# Sales ALL('Date'[Date])], a value of 3. There is one slicer per dimension: Date (with the second of the month selected) and Product (with ProductIDs 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're 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 5-15. A report showing the count of rows of the Sales table for all dates

The third section in the report shows similar content, but for measure [# Sales ALL('Sales'[Date])], and it 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 3.

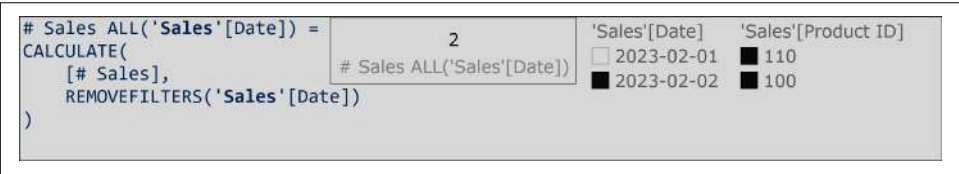


Figure 5-16. A report showing the wrong count of rows of the Sales table for all dates

This measure shows an unexpected value because of an optimization in the storage engine of Power BI/Analysis Services tabular. When the `REMOVEFILTERS()` function kicks in, the Sales rows with ProductID of 100 are 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 doesn't 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. It affects all kinds 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 applied directly on the fact table are in place; filters via dimension tables aren't affected by this optimization.

That's the long answer for why you shouldn't 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 this: 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 start with a single table and discover that you want to change it to a star schema later, doing so would break all of your existing reports based on the model.

On the other extreme, don't put everything into a single table, but fully normalize the data model to avoid redundancy. This isn't such a great idea for a data model in Power BI, which I explain next.

Normal Forms

Power BI is rather flexible about what kinds 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 for the report users to understand because information from even the same entity is spread out over multiple tables. And all the different tables need to be joined according to their filter relationships. Joins are expensive, leading to slower reports. Normalized data models are optimal for application databases, but not for analytical databases.

Again: if you start with a normalized data model and later discover that you want to change it to a star schema, it would break all the existing reports based on this model. Better start with a dimensional model up front.

Dimensional Modeling

You don't need to believe that dimensional data modeling is way better than all the other data modeling 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 on every level. 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's no need to actually store the data in a dimensional model form in the data source. For example, Bill Inmon (mentioned in [Chapter 1](#)) recommends storing 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 can either be derived from a normalized schema with the help of DAX, Power Query, or SQL. In this book, I teach all the necessary steps in all three languages—so no excuses anymore!

There are some important questions to ask your report users. How much 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 monthly basis? The answers to these questions define the data model's *granularity*.

Granularity

It's important that your fact table's granularity matches the primary keys of your dimension tables so you can create a filter relationship between them with a one-to-many cardinality. [“Budget” on page 135](#) gives an example of a case where new information needs to be added to an existing data model (the budget) that has a different level of detail: the budget is available only per product group, 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. [“Budget” on page 135](#) will also explain how to 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 probably not fit directly into a dimensional model. Luckily, we have tools available to first extract and transform, and then load the data into Power BI. The next section covers on these challenges.

Extract, Transform, Load

ETL isn't done via the Model view; you can use DAX, Power Query/M, or SQL to achieve this. Parts [III](#), [IV](#), and [V](#) dive into those languages and makes them your tools to extract and transform the data as needed.

Read on to learn about a special kind of transformation necessary to implement slowly changing dimensions, which isn't done in the Model view.

Key Takeaways

In this chapter, I matched the basic concepts of data modeling with features available in Power BI and Analysis Services tabular. You learned some key concepts:

- The Model view (in Power BI) and the Diagram view (in Visual Studio) give you a graphical representation of the tables and their relationships 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 column 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 is discussed in [Chapter 10](#).
- The cardinality of a relationship is represented by “1” or “*” (many). You can create filter relationships of the types one-to-many (the most common), one-to-one, and many-to-many. Many-to-many relationships can be created unintentionally when 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 either be propagated from one table to another or in both directions. Bi-directional filters risk making a data model slow. Situations could arise in which you can't add another table with a bi-directional filter when it would lead to an ambiguous model.

You now know how important a data model is in Power BI. [Chapter 6](#) shares practical knowledge of how to shape it into a dimensional model.

Building a Data Model in Power BI

Because Power BI is a data model-driven tool, it's important to ensure the information you display and interact with is modeled correctly. In this chapter, you will learn pros and cons of building a data model in Power BI. You will learn “how-to” techniques in DAX, Power Query, and SQL in Parts [III](#), [IV](#), and [V](#). Here, I talk about the principles and options in the Model view of Power BI Desktop.

I begin with a short recap on normalizing and denormalizing before broaching calculation writing. As you will see, certain types of calculations can't be done before loading the data into Power BI but only by defining the formula inside of Power BI.

Power BI can do common calculations for you without a specified formula. It's not a good idea to depend on this behavior, though. Always explicitly write even simple formulas.

I recap the importance of having a dedicated Date (and maybe Time) dimension in your Power BI data model. You will learn two ways of modeling role-playing dimensions and that slowly changing dimensions need to be modeled outside of Power BI (in a physical data warehouse layer). I end this chapter with a description of how to define and use hierarchies.

Let's begin with the most important part: normalizing and denormalizing.

Normalizing and Denormalizing

I introduce normalizing fact tables and denormalizing dimension tables to transform any given data model into a star schema in [“Data Model” on page 75](#). In the “Normalizing” and “Denormalizing” sections of Chapters [10](#), [14](#), and [18](#), I cover actual techniques for achieving this task in DAX, Power Query/M, and SQL. But in this chapter, I demonstrate different modeling approaches and their effects in Power BI.

I want to introduce Adventure Works, a fictitious sports article retailer that makes the majority of its revenue selling bikes on three continents both via resellers and directly through its web shop. Many examples in this book are based on this company's data warehouse (database AdventureWorksDW), mainly the tables FactResellerSales, DimDate, DimSalesTerritory, and DimProduct:

FactResellerSales

A table with sales made via resellers

DimDate

A table containing one row per day of a calendar

DimSalesTerritory

A table containing the sales regions, grouped into countries and continents

DimProduct

A table containing the goods, whose rows are assigned to DimProduct Sub category and DimProductCategory, to shape a product category hierarchy

For the following demonstration, I inflate the FactResellerSales table of the database AdventureWorksDW so it contains 18m rows (instead of the original 60,000 rows), and then I added this table and DimDate, DimProduct, DimSubcategory, DimProductCategory, and DimSalesTerritory and created three models:¹

- The model where I merge all information into one single table containing all the information has a size of 200 MB (OBT).
- If I keep the tables as they are to form a snowflake schema, the model size is 84 MB, including 704 KB for relationships.
- When I denormalize the three product-related tables into one, but keep all the other tables to form a star schema, the model size is, again, 84 MB with only 656 KB for relationships.

You might think that 200 MB is nothing to worry about, and I'd agree. Any laptop (and server, of course) will easily handle a 200 MB database. Any report on such a model will be fast enough. But that isn't the point. The point is that, by bringing the Adventure Works data model into a proper shape, you can reduce the size by half or two-thirds without losing information. That means you can use your existing hardware for hosting double or three times the information. Opening the .pbix file on your local machine will be faster. You can keep your premium subscription almost three times as long before you need to upgrade to the next bigger one. And so forth.

1 The examples in this section use files *Single Table.pbix*, *Snowflake.pbix*, and *Star.pbix* from the book's GitHub repository.

The single-table model doesn't spend any bytes on storing information about relationships because there are no relationships. The difference between the star and the snowflake schema is only a few kilobytes. While this difference is not impressive in this example, it again shows that reports built on a snowflake schema tend to be slower than reports built upon a star schema; some filters will need to traverse a longer distance over more tables.

Long story short: it's important to find the optimal compromise between normalizing and denormalizing your data model. The good news is that you don't have to invent something new. You can rely on a concept that has proved its usefulness over the past decades: a star schema, where you normalize all your fact tables and denormalize all your dimension tables.

The other example I like to use is a single Excel file provided by Power BI as a demo: *Financials*.² It contains the following columns:

- Segment
- Country
- Product & Manufacturing Price
- Discount Band
- Units Sold, Sale Price, Gross Sales, Discounts, Sales, COGS, Profit
- Date, Month Number, Month Name, Year

This is the classic example of OBT: all information is joined into a single table—which is less than optimal when it comes to Power BI (see “[A Single Table to Store It All](#)” on page 103). Again, I took the time to transform this table into different models. Here are some different approaches and their outcomes:

- Keeping the single table as it is results in 5.5 MB of total model size.
- Adding dimension tables (for Segment, Country, Product, Discount Band, and Date). The added tables occupy 5 MB, resulting in a 10 MB total model size.
- I add just one table (which contains all combinations of dimensional values) and a combined business key (where I concatenated all business keys, but could remove the single business keys from the original table). This two-table version of the data model occupies only 319 KB! Many thanks to Ana María Bisbé York for bringing my attention to this solution.

² This section's examples use the files *Financials Dimensional Model Surrogate Key.pbix*, *Financials Dimensional Model.pbix*, *Financials Filter Dimension Surrogate Key Measures.pbix*, *Financials Filter Dimension Surrogate Key.pbix*, *Financials Filter Dimension.pbix*, *Financials OBT Measures.pbix*, and *Financials OBT.pbix* from the book's GitHub repository.

This two-table version is what Ralph Kimball calls a *Junk Dimension*. As happens so often, I like Mr. Kimball's ideas but not their names. I don't think report users would be eager to add a column to a report that comes from a table named *Junk Dimension* or just *Junk* for that matter. I therefore name a table that contains the combination of dimensions just *Filter* instead. This concept works very well for dimension tables, which don't really have a lot attributes (like in the case of the *Financials* example, where every dimension, except for the product table, which also contains the *Manufacturing Price*, only has one attribute, which is simultaneously the table's key).

Again, look at this approach as another tool when it comes to finding the optimal data model. Build different proofs-of-concepts, before you head into one direction. The mileage for every concrete data model may vary, due to the characteristics of the database engine behind Power BI. What works well in one situation does not necessarily work well with different data.

Calculations

Calculations can be done in either DAX, Power Query, SQL, or any data source. If the result of a calculation is not additive (see the following list), then the calculation must be done as an *explicit measure* in DAX in order to achieve a meaningful result.

You (or your colleagues) can add calculations at several possible steps:

Adding a column in the data source

Using a formula in an Excel file, adding a column to a table or view in a database, etc.; you'll learn how to add calculations in relational databases in "[Calculations](#)" on page 376. The result of such a calculation is available to everybody with access to the data source.

Adding a custom column in Power Query

If I need to persist the result of a calculation and cannot do so in the data source, then Power Query is my next best option. The result will only be available inside this Power BI data model (and to everybody with access to the data model), and you will learn how to in "[Calculations](#)" on page 273.

Adding a calculated column in DAX

If you feel more confident in creating a column in DAX instead of Power Query, than you will choose this option.

Creating a measure in DAX

For semi-additive and non-additive calculations, a calculated measure written in DAX is the only option.

It's a good idea to add a calculation as early as possible in your stream of data (as far upstream as possible). If you create a calculation in Power Query or DAX, the result can only be used within the report (or reports built on top of this Power BI semantic

model). If you add the calculation in the data warehouse, other reports, models, and tools (and therefore a broader range of users) will benefit from the calculation as well.

Keep in mind, though, that only additive calculations can be calculated in the data source, Power Query, or as a calculated column. Semi- and non-additive calculations must be calculated as measures (and therefore in DAX, as laid out in “[Calculations](#)” on page 190).

But there is another option: numeric columns in a data model have a property called *default summarization*, as shown in [Figure 6-1](#). There, you choose one of the following options:

- Don't summarize
- Sum
- Average
- Minimum
- Maximum
- Count
- Count (distinct)

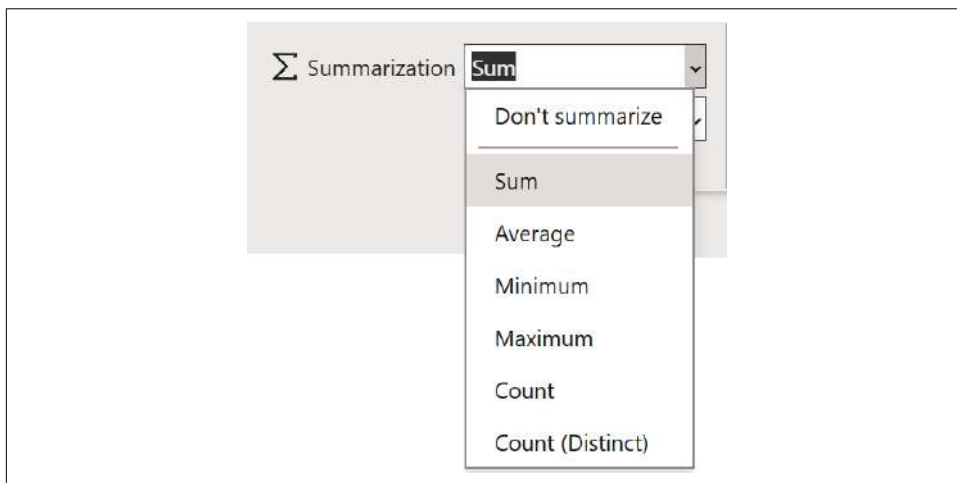


Figure 6-1. Default summarization

A similar list of options is available within visuals, as well (see [Figure 6-2](#)), allowing you to keep the default or define a different calculation for the scope of this visual. It allows the same options, plus standard deviation, variance, and median as well.

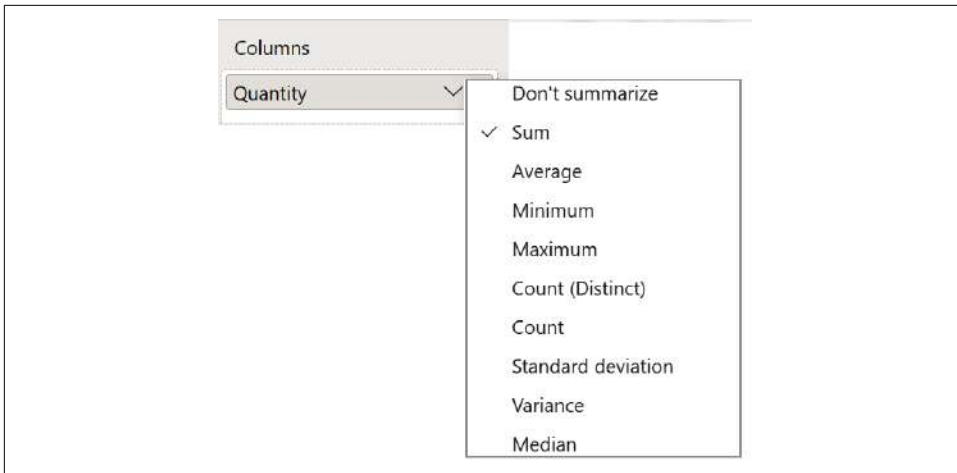


Figure 6-2. Visual summarization

This property is clearly aimed toward Power BI beginners: even without learning DAX, Power Query, or SQL, you can create (simple) calculations. Through this setting, Power BI creates an (implicit) measure for you, by applying the chosen calculation on top of the column. This implicit measure is not visible anywhere—you can neither view nor edit it.

But first, you, dear humble reader, are not a beginner anymore. And second, this feature has certain drawbacks:

- *Default summarization* doesn't work in all tools consuming a data model created in Power BI (in Excel, this was first added in 2023, for example). In general, tools accessing the semantic model with the MDX query language will ignore this setting. As a consequence, you won't be able to show the aggregated value of a numeric column, even when with default summarization switched to something besides "Don't summarize."
- If you start using calculation groups, all implicit measures are voided. Visuals in which you used an implicit measure, will show an error message. You need to manually fix those visuals by first explicitly creating the measures and then exchanging the columns with the measures.

You never can be sure if your data model's users will use Excel (or any client that uses MDX instead of DAX to query), or if you'll want to introduce a feature in the future that might prohibit implicit measures (e.g., calculation groups). It's best to hide numeric columns and define explicit measures instead. ["Calculations" on page 190](#) covers how to do this.

On top of that, Power BI automatically sets a default summarization mode for every numeric column, either by setting it to Default (which sums the value) or Count (for the primary and foreign key columns used in filter relationships). The Default can lead to confusing numbers in your report for columns like Year or Month number; it doesn't make sense to add such numbers. The Count can be meaningful, but I'd prefer to create an explicit measure with a good name like Count of Customers instead.

Therefore, it's important to take the time to scan through all numeric columns in your data model and perform some assessments:

- Columns whose values should be aggregated should be hidden. Create an explicit measure. In some cases, it might make sense to create more than one explicit measure (one for the sum, another for the average, for example).
- For columns whose values should not be aggregated, default summarization must be set to "Don't summarize."

While you're scanning through your columns, use the opportunity to take a look at other properties as well. The following are not directly connected to calculations, but influence the behavior of the column's values in a visual:

Data type

Data type defines what kind of values can be stored in this column (and, therefore, influences the storage format).

Format sets

Format sets how the value is shown. You can choose one of the variety of options or **use a custom format string**.



The Format property doesn't change the data type. This can be a problem for columns of any data type, but especially with columns of data type Date/Time. If you change the (display) format to only show the date portion but keep the data type as Date/Time (instead of Date), then the time portion will still be stored in the model. The column will occupy unnecessary space in memory and on disk.

You might get into trouble when you create a filter relationship between columns with Date/Time and Date properties when the former indeed contains timestamps. When the filtering is propagated, the Date column's time will be midnight, while the Date/Time-type column's time will be the actual timestamp—and no related columns will be found.

Data category

Some visuals will react to the “data category” property and show the content in a certain way. For example, if you categorize something as an address or city, then Power BI will choose a map visualization by default.

Sort by column

Typically, you want the month names not ordered alphabetically (which would show “April” first and “September” last) but ordered by month number. The option “Sort by column” enables exactly this: by choosing column `Month Number` as the “Sort by column” of `Month Name`, all requests to order by `Month Name` will automatically be changed to order by `Month Number` instead—and “January” will be the first and “December” the last month, as is usually desired.

Time and Date

Having a `Date` dimension is crucial for many data models; therefore, you need to take extra care of this dimension table.

Turning off Auto Date/Time

Power BI can automatically create a date table for you.³ It will contain four columns: `Year`, `Quarter`, `Month`, and `Day`, which are grouped nicely in a hierarchy. In [Figure 6-3](#), you can see that I added a `Date` column with the option `Date Hierarchy` enabled. You can choose `Date` instead, to remove the hierarchy and show a single column containing the date instead. If you do not want to show the `Quarter` for example, you can just remove it by clicking on the X to the right of `Quarter`.

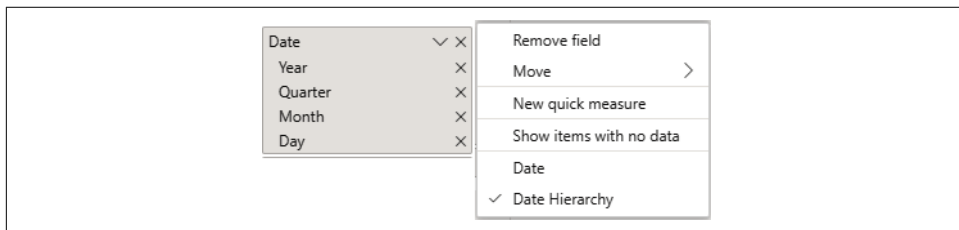


Figure 6-3. Auto-generated date hierarchy

In [Figure 6-4](#) you see the content of the four hierarchy levels of the auto-generated date table (and column `Amount`).

³ The examples in this section use the [Auto date-time.zip](#) file from the book’s GitHub repository.

Sales (Date)				
Year	Quarter	Month	Day	Amount
2023	Qtr 3	September	1	23000
2023	Qtr 3	September	2	10000
2023	Qtr 3	September	3	15000

Figure 6-4. Auto-generated date hierarchy shown in a table visual

Unfortunately, this auto-generated date table comes with a bunch of disadvantages:

- You can't add, change, or remove columns from the hierarchy.
- Power BI will create such a table for *every* column of data type Date or Date/Time in your data model. This sounds like an advantage but is not: the memory footprint of your data model will increase unnecessarily.
- If you later turn this feature off or create a relationship between this column and your date table, it will change your visuals or even break them.

If your report creators won't ever use the `Quarter` column in reports, they need to add the hierarchy and remove the column `Quarter` from the visual every single time. If they need the `Month` in a different format (like `Sep, 09`, or `September 2023`) or expect the `Day` column to contain the full date or include the weekday, you can't use the auto-generated date table but need to create your own date table (which you will learn in the "Time and Date" sections in Chapters 10, 14, and 18).

Power BI will automatically detect a date range by looking at the earliest and most recent entry and then create a (hidden) date table, covering the beginning of the first year (January 1) until the end of the last year (December 31) for each and every column of data types `Date` and `Date/Time`, even when you don't intend to use this hierarchy for this column. This can lead to multiple huge tables in your data model with a widespread range of dates. For example, if you load the birth date of people into your data model, the date table created for just this column will easily cover many decades; if your data source uses placeholders like `January 1, 1900`, `December 31, 9999`, or `January 1, 0000`, the automatically created date tables will easily occupy a remarkable amount of space in your Power BI file and in memory. These auto-generated date tables are hidden in Power BI Desktop; you need third-party tools to see them. Figure 6-5 was created with DAX Studio and shows a list of these tables (with prefix `LocalDateTable`).

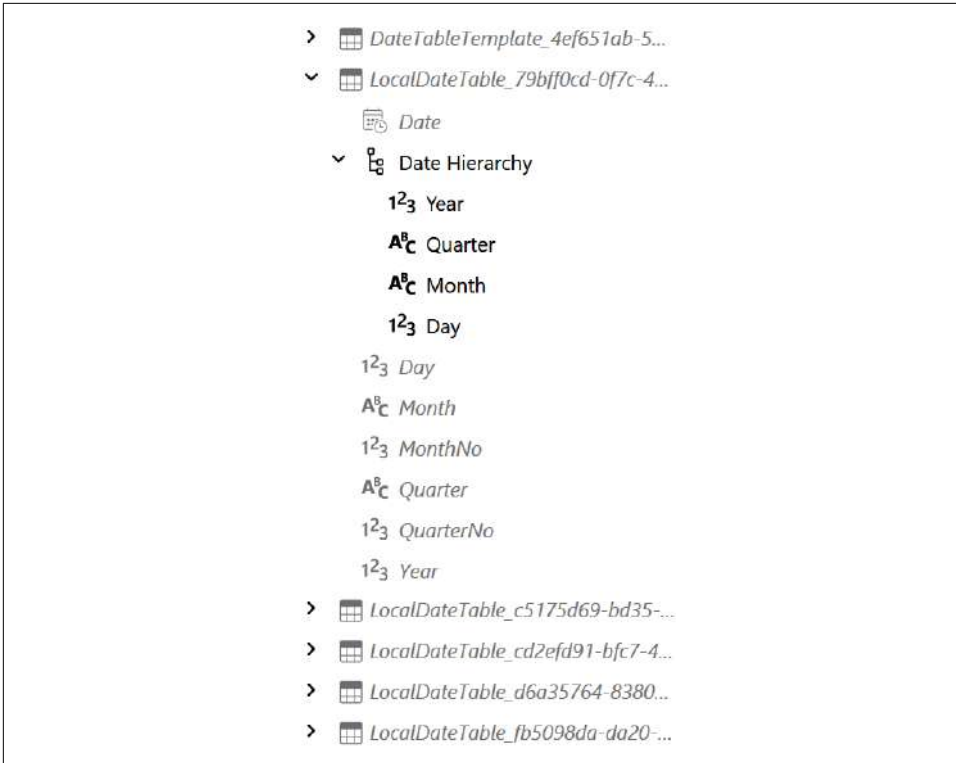


Figure 6-5. Auto-generated date tables shown in DAX Studio



According to [the official documentation](#), Power BI supports date ranges from January 1, 1900 to December 31, 9999. I could, however, successfully create calculated columns in DAX starting in the year 0100 (not that I ever need to, but I was curious). Values with a year earlier than 0100 (or subtracting days from 0100-01-01) are automatically moved into the 20th century in DAX: 0000 is interpreted as 1900 and 0099 as 1999. You can't create negative years (e.g., for years in the BC era).

VertiPaq Analyzer, available in DAX Studio, can also show you the size of these tables. As you can see in [Figure 6-6](#), the five tables with prefix LocalDateTable contain between 365 and 3,615,900 (!) rows, occupying something between 60 KB and 269 MB. The biggest table was created for a column containing dates between January 1 of year 100 and December 31, 9999. This might be an extreme example, but demonstrates what can happen below your radar when you do not turn the auto date/time setting off.

Name	Cardi... ↓	Total Size
▶ LocalDateTable_cd2efd91...	3.615.900	269.755.696
▶ LocalDateTable_79bff0cd...	1.096	115.364
▶ Date	731	51.080
▶ LocalDateTable_fb5098da...	730	84.792
▶ LocalDateTable_d6a3576...	365	61.932
▶ LocalDateTable_c5175d6...	365	62.064
▶ Sales	4	9.224
▶ DataTableTemplate_4ef65...	1	36.316

Figure 6-6. VertiPaq Analyzer exposes the size of the auto-generated date tables

If you turn the auto date/time setting off after somebody has included the hierarchy in a visual, the visual will change. Figure 6-7 shows how Figure 6-4 changed.

Sales (Date)	
Date	Amount
2023-09-01	23000
2023-09-02	10000
2023-09-03	15000

Figure 6-7. The same visual, but with auto date/time turned off

On top of that, referencing one of the four parts of the auto-generated hierarchy (Year, Quarter, Month, and Day) in a DAX calculation has a special syntax, like `Sales[OrderDate].Year`. The Year isn't referenced as a column but as a variation of the Date column.

If you change your mind and create your own Date table, it will void all measures that use this special syntax, which can easily happen if you implement time-intelligence calculations (see “Calculations” on page 190). Working measures are suddenly voided, displaying an error message like this: “Column Reference to Date in table Sales cannot be used with a variation Year because it does not have any.” Your data model's users will see a gray box with an error message instead of the visual they're expecting (Figure 6-8), announcing a problem with a measure. This might not only be the case with measures defined in the data model (which you have under control and can fix). Measure definitions inside reports and queries can also be affected outside your data model, created by users or their tools.

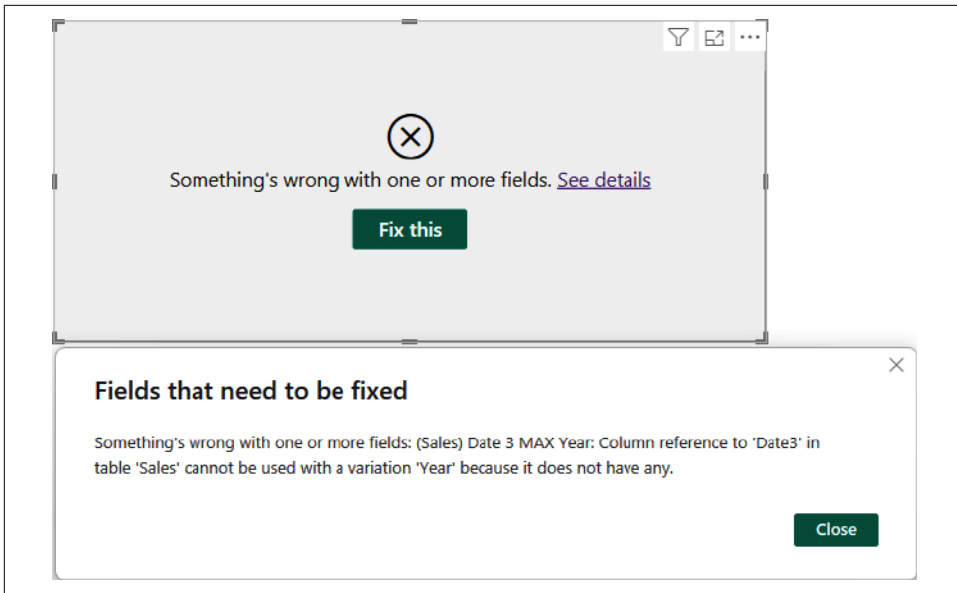


Figure 6-8. The same visual, but with auto date/time turned off, containing a voided measure

Do not click “Fix this” because it will “fix” the visual by removing all voided columns or measures. That’ll get rid of the error message but make it difficult to find which measure to add back after it’s corrected. Correcting the measure isn’t too hard: you need to change parts of the code from something like `Sales[OrderDate].Year` (which references the variation of the fact table’s date column) to something like `Date[Year]` (which references the Year column of your Date table).

I consider it dangerous not to disable auto date/time *from the beginning, when you start* creating a data model. Turn this feature off for individual files via File → Options → “Options and Settings” → Current File → Data Load → Time Intelligence, as shown in [Figure 6-9](#).

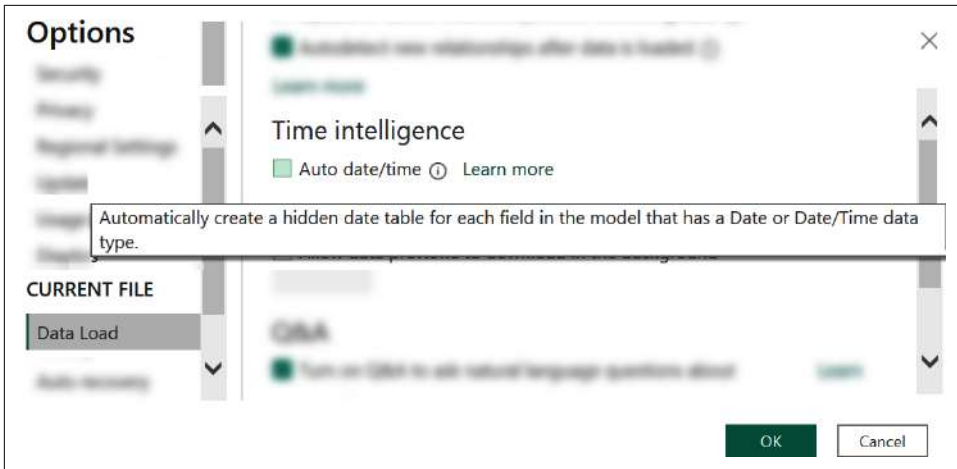


Figure 6-9. Current file’s data load options for time intelligence

To make sure this feature is turned off by default on your computer (which I strongly recommend), go to File → Options → “Options and Settings” → Global → Data Load → Time Intelligence. [Figure 6-10](#) shows this setting.

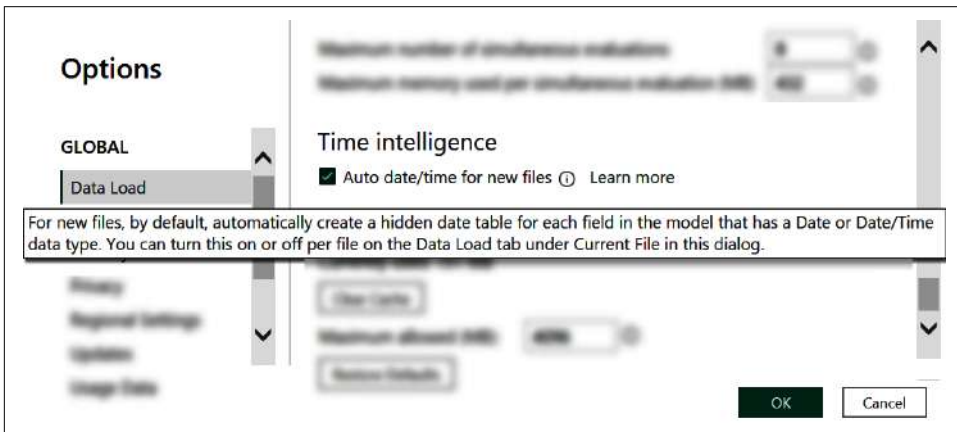


Figure 6-10. General data load options for time intelligence

In [Chapters 10](#), [14](#), and [18](#), you’ll find scripts to create your own date table. These scripts give you full control over the columns and their format.

Marking the Date Table

Another thing you shouldn’t overlook is explicitly marking your date table as a date table. I recommend getting into the habit of doing this for all your date tables, even though it’s not always necessary, because it won’t do any harm and it’ll be easier to

remember if you do it all the time. You must mark the date table when *both* of the following conditions are met:

- You (or your report creators) want to use DAX’s built-in time intelligence functions.
- You aren’t using a column of data type Date or Date/Time to create the filter relationship. (In data warehouses, it’s best practice to use a column of data type integer as the primary key of the date table, for example.)

If both conditions are met, DAX’s time intelligence calculations won’t work properly unless you explicitly mark your date table. Luckily, it’s very easy to do:

1. Select the date table.
2. Choose “Table tools” → “Mark as date table” in the ribbon.
3. Select the date column from the list box in the dialog box.

The “Date column” dropdown will only list columns of data type Date or Date/Time.

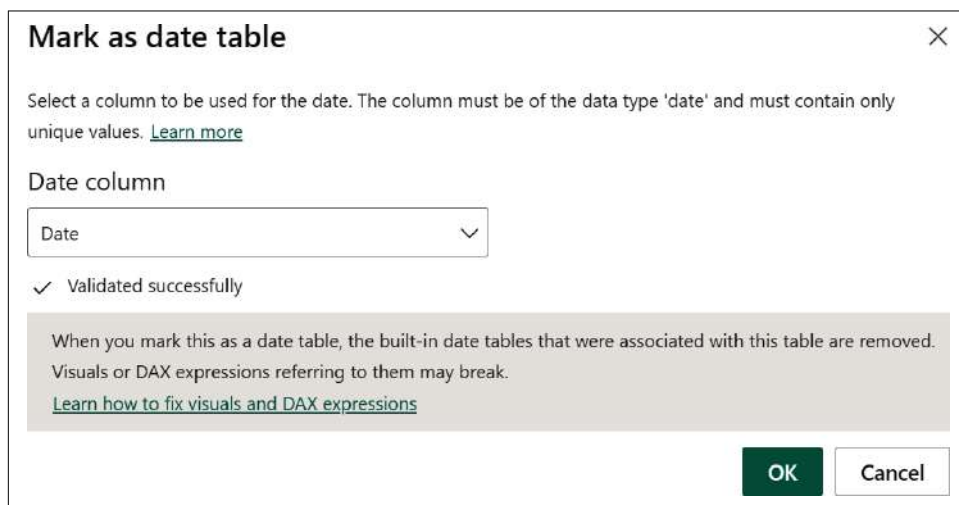


Figure 6-11. Marking the date column of your date table

Now you’ve made sure that your data model has an explicit date table. In some data models, however, more than one meaning is possible for the shown or selected date. The report consumer might filter on dates when the goods have been ordered or on dates when the goods have been shipped. In this case, the Date table might play more than one role. The next sections cover two different solutions for this problem: role-playing dimensions and slowly changing dimensions.

Role-Playing Dimensions

A dimension in a data model can play more than one role. It's most typically seen with the date dimension, although multiple roles are seen elsewhere as well. Let's assume that the Sales table stores both an Order Date and a Ship Date column.⁴ You have two options to make this work in the data model:

Create two relationships between the Date table and the Sales table

One filter relationship uses the Ship Date column of the Sales table. The second one uses the Order Date column. This is shown in [Figure 6-12](#). You'll need to use DAX to create dedicated measures per relationship in which you activate the inactive relationship because only one of the two relationships can be marked active. "[Role-Playing Dimensions](#)" on [page 206](#) goes over all the details. You'll end up with only one dimension table, but several measures.

Duplicate the Date table and create two relationships from there

Name one copy of the Date table Order Date and the other Ship Date. Create one relationship between the Order Date table and the Sales table and another between the Ship Date table and the Sales table (shown in [Figure 6-14](#)). You don't need to build additional versions of measures because you have only active relationships. In the end, you'll have several versions of the dimension table, but only single versions of the measures.

Let's explore the differences between the two options. Both options cover slightly different use cases and requirements:

[Figure 6-12](#) shows a model with two tables, Date and Sales, and the two relationships between them. One is based on the Sales table's Ship Date column and one is based on the Order Date column. One of the two lines representing the filter relationship is continuous and the other is dashed. The continuous one is an *active relationship* and the dashed lined represents an *inactive relationship*.

Only one relationship (filter path) between two tables can be active. You can only add inactive relationships because, if you had two active relationships, the model would be ambiguous. Should a filter on the Date table's Year column filter all the sales that *took place* this year or that *were shipped* this year? Should the filter be applied to both so we only get to see sales that were ordered and shipped in the same year? To avoid ambiguity, Power BI will only respect the active relationship—you need to choose which relationship should be active. If you want to make a different relationship

⁴ The examples in this section use the *Date role-playing.pbix* and *Datamultiple.pbix* files from the book's GitHub repository.

active, then you first have to deactivate the active one. At one point in time, only one relationship path can be active.

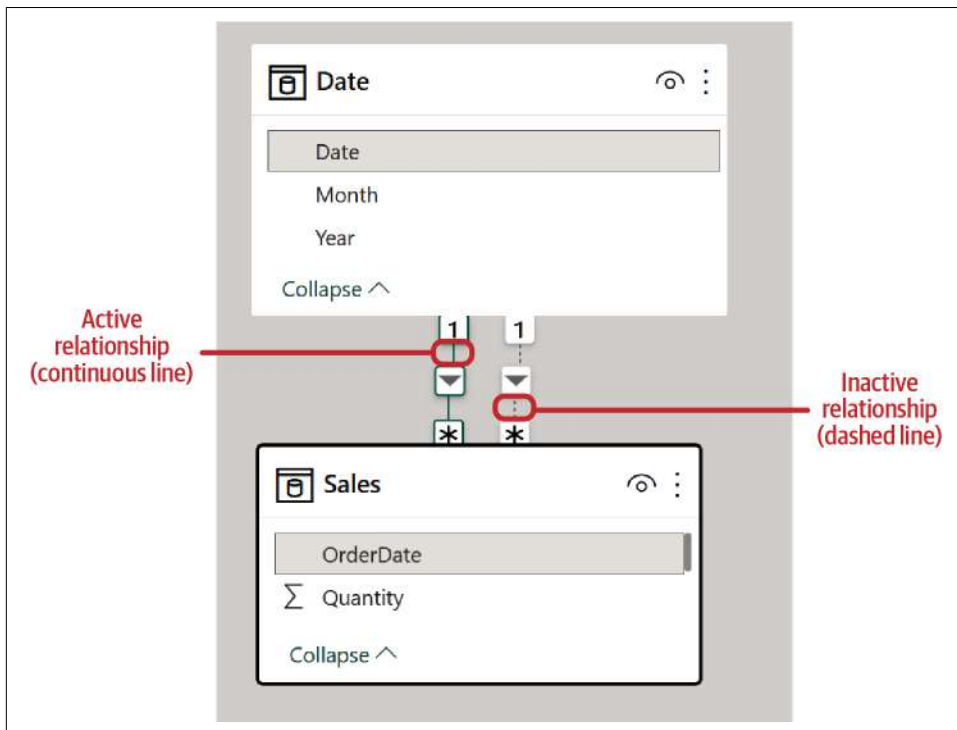


Figure 6-12. The Date table and the Sales table have two relationships

Why would you create additional relationships in the data model if they can't be active? There are a few reasons:

- Creating inactive relationships will make it transparent that this is a valid way of propagating a filter from one table to the other.
- The storage engine inside Power BI will add extra (hidden) structures/information to the data model to make filter propagation based on this relationship faster (compared to when you don't define a relationship at all)
- You can also activate an inactive relationship inside a calculation done in DAX.

The `USERELATIONSHIP` function can be used to activate an inactive relationship. One of the solutions described in [“Role-Playing Dimensions” on page 206](#) adds only one generic Date table to the data model. You then decide inside the formula of a measure if one or the other relationship should be used for the calculation of this measure. `USERELATIONSHIP` will temporarily activate the chosen relationship (and

temporarily inactivate the existing active relationship). For example, you can create one measure for the default relationship and name it [Order Quantity] and then create another measure with the same formula where you activate the relationship on the Ship Date column to calculate the [Ship Quantity].

The advantages of this approach are that you only load one version of the dimension table (e.g., Date) and that you can use this dimension in visuals where you can then compare the same value/measure side-by-side based on each of the two role-playing options. For instance, in the USERELATIONSHIP example, you can compare the [Order Quantity] against the [Ship Quantity] side-by-side based on dates, as shown in [Figure 6-13](#).

I intentionally used USERELATIONSHIP in both of the measures, although it would be sufficient to use it only to activate the inactive relationship (the one on [Ship Date] in my example). I do this as a fail-safe in case somebody changes which of the relationships are (in)active.

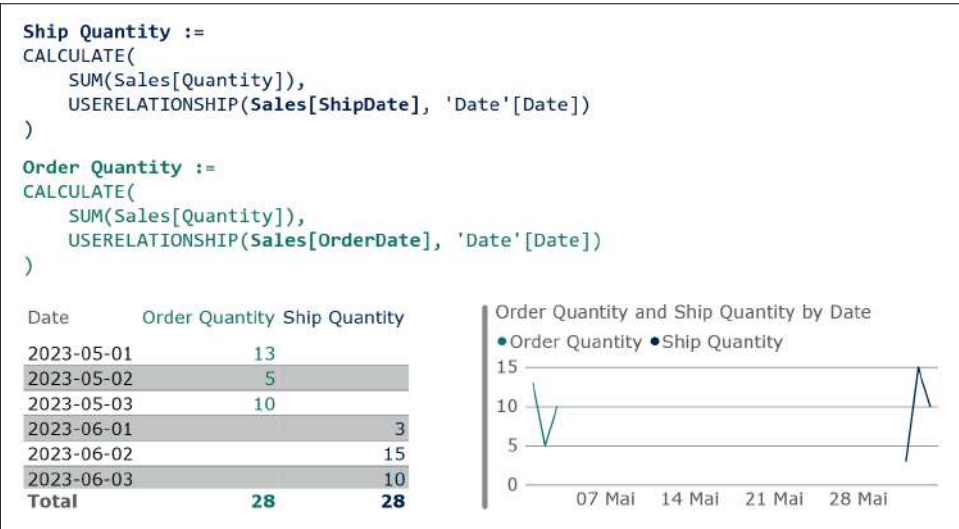


Figure 6-13. A single Date table and two measures

Alternatively, you can implement a solution that works without DAX. Then you need to duplicate the Date table under different names (you can learn how to do this in [Chapters 10, 14, and 18](#)).

In [Figure 6-14](#), you see this implemented for the previous example. You'll end up with several dimension tables (one per role, e.g., Order Date and Ship Date) that have each an active relationship with the Sales table. For example, the Order Date table's Order Date column will have a filter relationship to the Sales table's Order

Date column. Similarly, the Ship Date table's Ship Date column will have a filter relationship to the Sales table's Ship Date column.

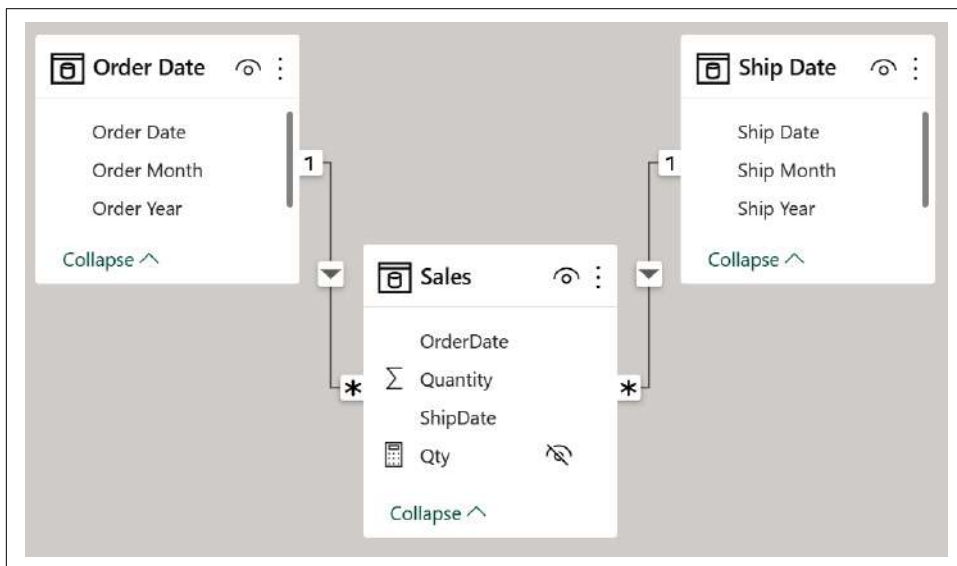


Figure 6-14. A model with two dedicated date tables

Following this approach, the report user can choose from two filters for each, the Order Date and Ship Date, and use one single measure. You can also create a matrix visual showing when goods have been ordered and compare that to when goods have been shipped. Both dimensions filter the identical measure, as shown in [Figure 6-15](#).

Now you've learned that the difference isn't only whether you use DAX but that the two solutions give you different options in the reports. If you want to cover all use cases, you can, of course, build a model with three date tables (Date, Order Date, and Ship Date), four filter relationships to the Sales table (two from the Date table and one each from tables Order Date and Ship Date), and three measures to calculate the different versions of quantity.

In this section, I talked about when dimensions play different roles within a data model. Next, you'll learn how to keep track of changes for a dimension.

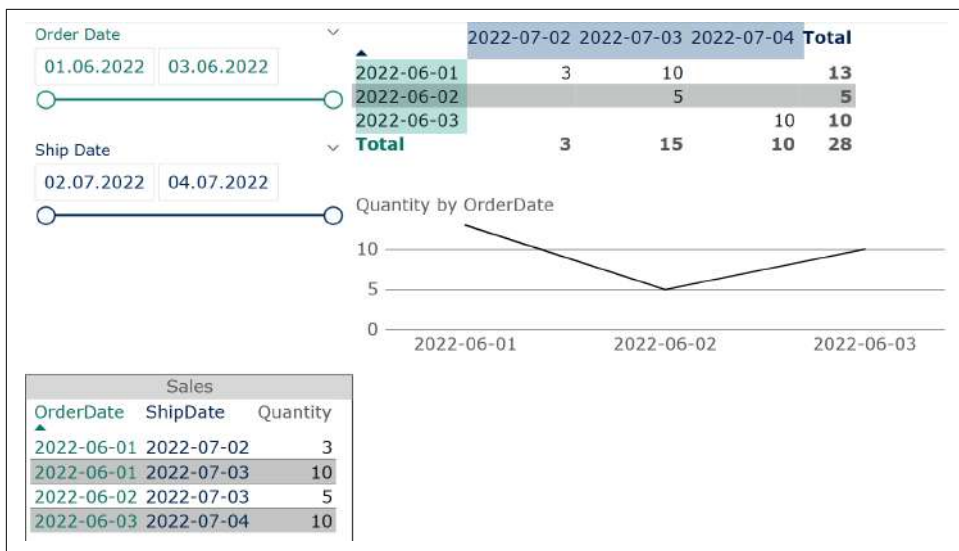


Figure 6-15. Two independent dimensions/filters and one measure

Slowly Changing Dimensions

To implement a solution for slowly changing dimensions, you need the possibility to discover changes and selectively update information in your analytical database. There is no way to compare the existing data with the new data during a data refresh in Power BI. You can only refresh a whole table, but not selectively certain rows in the table. Therefore, you need to implement slowly changing dimensions in a (physical) data warehouse.

If slowly changing dimensions were implemented correctly in the data source, there is nothing special to do in the Model view, except for creating the filter relationships. All the dimension tables will already track the historic changes. And the fact tables will refer to the primary key of the dimension's row matching the point in time of the fact table.

Your task in Power BI is just to create a filter relationship between the fact table and the dimension table on the dimension table's (surrogate) key column (like in [Figure 6-16](#), where the Product table and the Reseller Sales table are connected simply via the Product Key column).⁵ All rows in the fact table will always contain the surrogate key of the dimension's row version that was active at the point in time when the fact occurred. So, there's no need for a complicated logic to find the right

⁵ The examples in this section use the *Slowly Changing Dimensions.pbix* file from the book's GitHub repository.

version of the dimension table's row when joining with the fact table. An ordinary filter relationship will be sufficient.

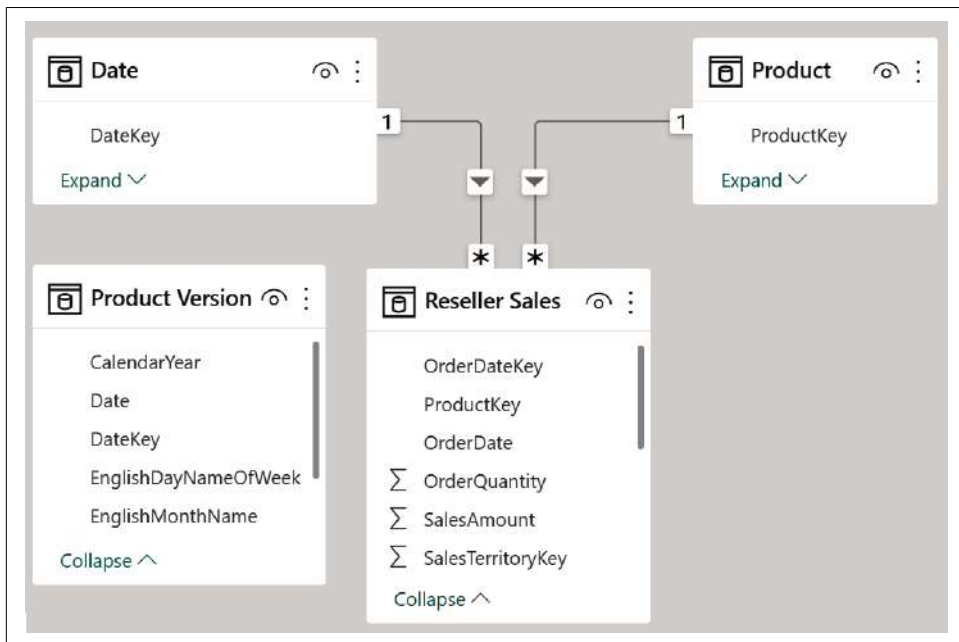


Figure 6-16. Filter relationship between the table Product and the table Reseller Sales

If your report users ask you to let them choose which version of a dimension to use, I have a solution for you. By default, the numbers for the actual year are displayed according to the dimension's current version, and the numbers for the previous year are displayed for the dimension's previous year's version. By introducing a detached table (see Product Version in [Figure 6-16](#)) containing dates or version numbers and some logic in a DAX measure, you can implement a model where the user can choose a version and override the relationship between the fact table and the dimension table. You'll learn to implement this case in [“Slowly Changing Dimensions” on page 207](#).

Many data models contain some sort of hierarchical data. This is not so true for Power BI, but it can be convenient for the report user if they can find predefined hierarchies in the data model.

Hierarchies

In *multi-dimensional cubes*, defining hierarchies is a very crucial step. *Analysis Services Multidimensional* (the predecessor of tabular models like Power BI) uses this information to (automatically) calculate and store aggregations of the fact table's data to speed up the queries. Lack of definitions (or mistakes) can result in slow queries.

In Power BI and the Analysis Services tabular model, the definition of hierarchies plays a more secondary role. They make creating reports easier because a user can add several columns with just one click—but the lack of definitions of hierarchies doesn't influence the size of the data model or the speed of the reports (at least not at the time of writing this book).

Nevertheless, the goal of data modeling is indeed making report creating easier. Therefore, you should still check your data model for hierarchies. In natural hierarchies, something is part of something bigger: a day is part of a month, which is part of a year. Upper Austria is part of Austria, which is part of the European Union.

An *organigram* shows who is reporting to who within an organization (and is an example for a *parent-child hierarchy*). Sometimes reports show things like the available colors per T-shirt size—this is not a natural hierarchy (most colors are available in all T-shirt sizes), but if this is a common request, I'd also define a hierarchy within Power BI for this case.

Power BI and Analysis Services tabular don't directly support parent-child hierarchies. You can add columns from within one table as a part of a hierarchy, but Power BI can't automatically find its way through, e.g., the `Manager ID` column of one row to get to the `Employee ID` column of another row and grab the information. The way around this is that we, the data modelers, have to dissolve the parent-child hierarchy by adding all information of the ascendant levels to every single row. We need to add a column with the CEO, a column for the VP level, and so forth, to the employee table. In the "Hierarchies" sections of Chapters 10, 14, and 18, I demonstrate how to create the content for those columns.

To make the report user's life easier (the ultimate goal of a data model), you can group columns together to form a hierarchy. This saves the user from finding every single column to add to a visual—they can instead add the hierarchy (and thus all columns within it). This feature is independent of the type of the hierarchy. The important part is that all levels that should form the hierarchy be separate columns within the same table. If the columns are spread out over several tables, you can't define a hierarchy. You must first denormalize the columns into a single table. If you

decide against denormalizing, the user can still add individual columns to a single visual to show hierarchical information.⁶

You define a hierarchy in the Model view:

1. Right-click a column to Create Hierarchy.
2. Choose “Select a column to add level,” as shown in [Figure 6-17](#).
3. Click “Apply Level Changes” when you’re finished to ensure that you don’t lose the whole definition.

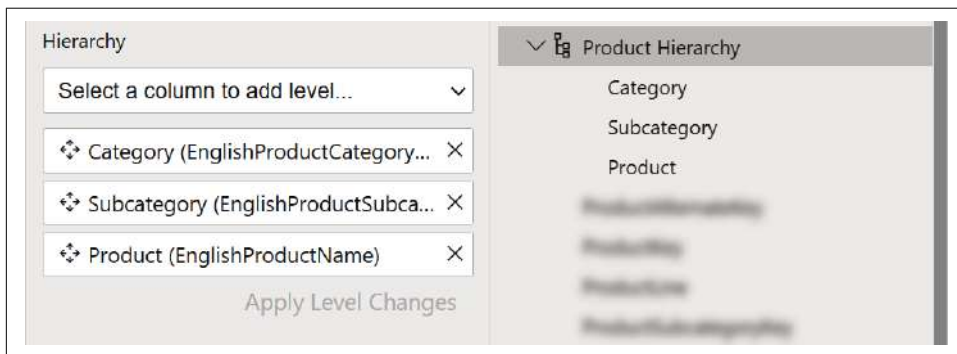


Figure 6-17. Definition of hierarchy “Product Hierarchy”

Key Takeaways

In this chapter, I laid out typical tasks that I’ve found I need to complete in almost every single Power BI data model I build. You learned the following:

- Bringing your data model into a star schema by denormalizing all dimensions and normalizing all your fact tables is the most optimal way to leverage data compression and report performance.
- Relying on implicit measures (via the default summarization property of numeric columns) is considered bad practice and limits the capabilities of your data model in some tools. It might compromise reports when you introduce calculation groups later.
- Taking care of the properties of your numeric columns is important (e.g., setting the default summarization to “Don’t summarize” or hiding the numeric columns for which you created an explicit measure). You also want to take care of the “Sort by column” option for some text columns (like month names).

⁶ The examples in this section use the [Hierarchies.pbix](#) file from the book’s GitHub repository.

- Auto date/time is turned on by default in Power BI Desktop, but it's considered bad practice to enable it because of its inflexibility and the potential size of the date dimensions created for you. Turning this setting off later might break the syntax of your measures or change the look and feel of existing visuals. That's why you should turn it off from the beginning.
- Make sure to understand both role-playing dimensions and apply one or the other (or both concepts) in your data model.
- Usually, you don't need to pay attention to slowly changing dimensions in your data model; all the logic is implemented in a persisted data warehouse. Only in rare situations will you face the requirement that the report user needs to choose the version of the dimension to show data based on that version.
- Hierarchies are of great importance in many data models. Natural hierarchies are easy to implement by fully denormalizing a dimension (which you should do anyway). Parent-child hierarchies need to be persisted into one column per level. As soon as you have the different levels as dedicated columns within one table, it's easy to add them to a hierarchy.

Now that you know how to build the basic parts of a data model in Power BI, it's time to dig into real-world use cases and how to solve them via the data model.

Real-World Examples Using Power BI

In Chapters 5 and 6, I introduce the basic data modeling features of Power BI and typical tasks to transform your data into a useful, performant, and easy-to-understand data model. Now it's time to take the next step and look at challenges that more advanced data models must face.

This chapter makes clear why Microsoft's decision to make Power BI a *data model-driven* tool (as opposed to a *report-driven* tool) was a brilliant decision. Instead of building transformations specific to the report you need to create, Power BI enables you to build a more generic solution, which can solve very complex challenges. All problems discussed in this chapter would be harder to solve in a report-driven tool.

This chapter's use cases are independent of each other. All demonstrate different advanced functionalities in Power BI. Mastering them will allow you to solve others problems as well. Here's a quick overview:

- The first use case (binning) demonstrates how to show a higher-level grouping of a value (e.g., small, medium, large) instead of the actual value, which is sometimes more helpful.
- In “Budget” on page 135, you'll learn about multi-fact data models and their challenges. The data model in this example will contain more than one fact table, as a budget is usually created on a different granularity level than the actual values.
- The available solutions to implement multi-language models didn't satisfy one of my customer's needs, so I designed and implemented one myself. I describe it in “Multi-Language Model” on page 139.
- In another case, I was faced with a problem—key-value pair tables—during one of my projects and developed a solution to handle them gracefully, which I share here.

- DirectQuery for Power BI semantic models and Analysis Services finally became generally available in April 2023. DirectQuery basically means that you don't import data into Power BI; Power BI *directly queries* the data source whenever a report page is shown or has a filter changed by a user. This feature helps tremendously in combining self-service BI and enterprise BI, as you will learn in “[Combining Self-Service and Enterprise BI](#)” on page 150.

Keep in mind that you still need to start from a star schema to solve advanced or challenging use cases. It can be difficult, even impossible, to solve challenging use cases in which you start from a data model that violates the principles of a star schema.

Binning

In this section, I discuss two options that should be familiar from “[Binning](#)” on page 58, and use the *Binning.pbix* file to demonstrate:

- A lookup table with all possible values
- A lookup table describing the value ranges of a bin

Lookup Table

If you have a lookup table containing a row for every possible row of the value plus the value to show as the bin, you only need to create a single-directed, one-to-many filter relationship to the value of the fact table. In [Figure 7-1](#), the Bin Table's primary key Quantity has a filter relationship with the Quantity column of the Sales table. Quantity here has the dual role of the actual quantity and the foreign key to the Bin Table. The latter role may look a bit unusual but has a good performance and is therefore more than acceptable.

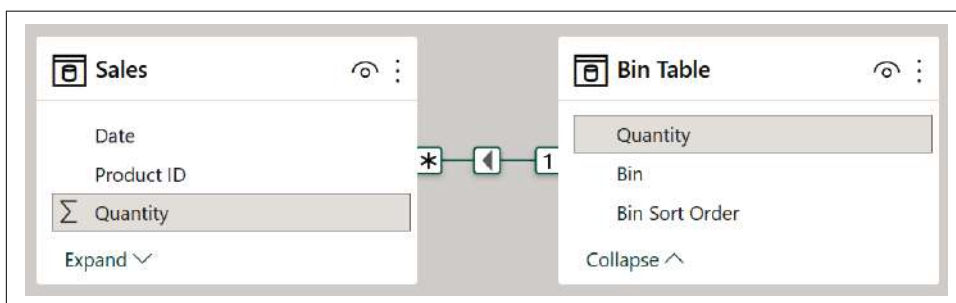


Figure 7-1. Relationship between the fact and the bin table

Range Table

If your lookup table contains not all distinct values, but the lower and upper range of the bin instead, you can't create the proper filter relationship in the Model view—Power BI only allows for equi-joins. You can (and need to) solve this with the help of DAX instead. **Figure 7-2** shows two unrelated tables. (See “[Binning](#)” on page 216 for more about this.)

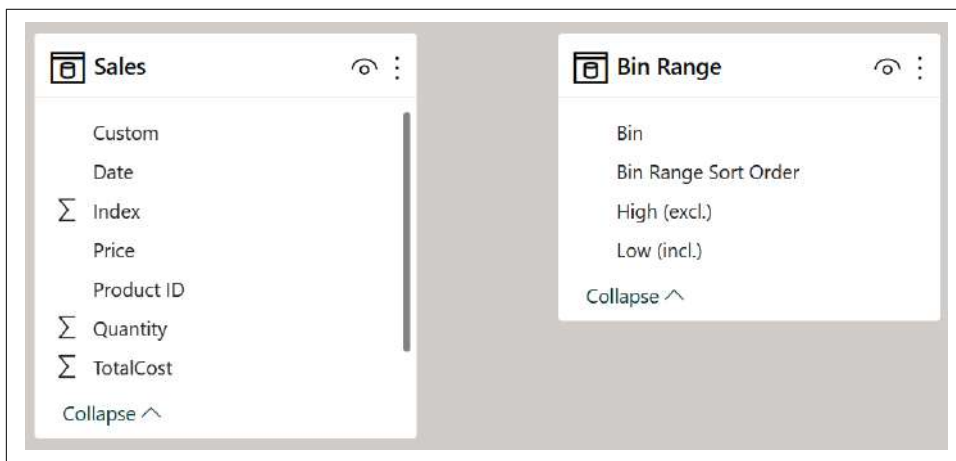


Figure 7-2. Relationship between the fact and the bin range table

In both cases, make sure to add a column that contains the sort order to the table containing the bins. Usually, people expect the bins to be ordered in a certain way (which might not be alphabetical). For example, low, average, high (instead of in alphabetical order—average, high, low). Click the bin's text column and select “Column tools” → “Sort by column” from the ribbon and select the column that contains the values for the proper order.

The next section is about when you need more than one fact table in your data model. It is a generic use case, and I will demonstrate it on the basis of a budget.

Budget

As a model-driven tool, Power BI allows for as many tables of any type to be added to the data model as you need to solve your business problem. Therefore, creating a multi-fact data model (with, e.g., one fact table for the actual values and another fact table for the budget) is no problem at all. The challenge is to find the best way to bring the budget table (which has a different granularity level than the table for the

actual values) into a relationship with the rest of the tables. So the budget use case is only a model for how to implement any multi-fact data model.¹

To solve this challenge, you have to choose between three potential solutions available in Power BI, which have their own advantages and disadvantages:

Keep Budget and Product tables

Because the budget is on, e.g., the product category level, but not that of the individual product, you don't create a relationship between the Budget table and the Product table at all (see [Figure 7-3](#)). You keep the Budget table and Product table disconnected. The solutions from “[Binning](#)” on [page 134](#) can be applied here too: using DAX to create the relationship on the fly where needed (see “[Binning](#)” on [page 216](#)).

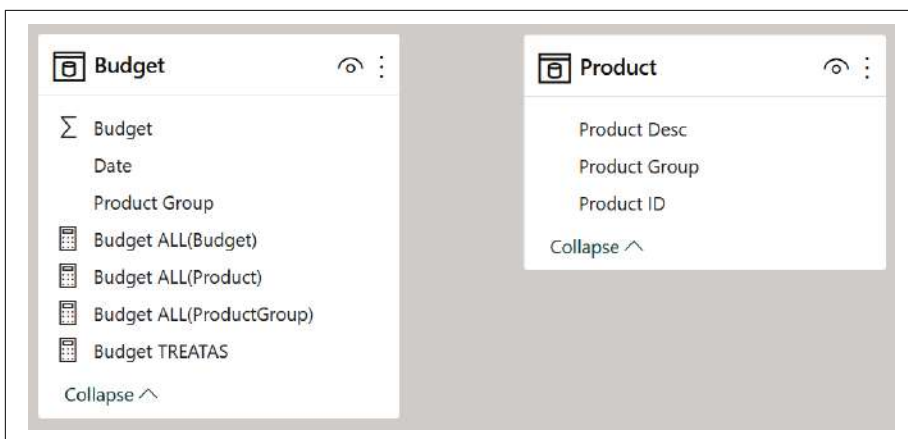


Figure 7-3. No relationship between tables Budget and Product

Create a relationship between the Budget and Product tables

Alternatively, you can create a relationship between tables Budget and Product (see [Figure 7-4](#)).

¹ The examples in this section use the [Budget.pbix](#) file from the book's GitHub repository.

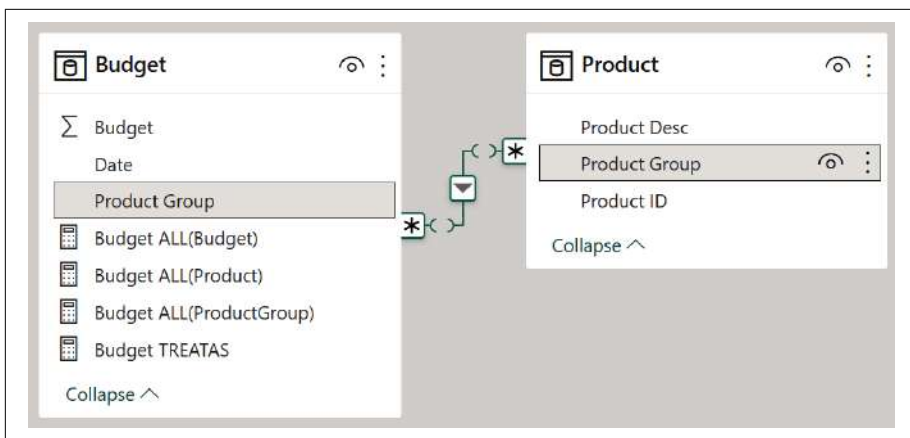


Figure 7-4. A many-to-many relationship between tables *Budget* and *Product*

After doing so, you'll receive the following message (see [Figure 7-5](#)):

This relationship has cardinality Many-Many. This should only be used if it is expected that neither column (Product Group and Product Group) contains unique values, and that the significantly different behavior of Many-Many relationships is understood. [Learn more](#)

It's only a warning, so you can successfully create such a relationship by clicking OK. It's important, though, that you fully understand (and accept) the consequences of such a limited relationship (see ["Relationships" on page 88](#)). One is that no blank row is shown to represent missing values—they're just skipped. When you follow all best practices, there should be no missing values anyway. Another consequence is that the function ALL in DAX won't remove a filter from related tables. In this example, that means that removing filters from the Budget table won't remove the filter on the Product Group column of the Budget table.

Make sure to make this relationship single-directed (Product filters Budget) to avoid an unnecessary bi-directional filter.

Create relationship

×

Select tables and columns that are related.

Budget

Date	Product Group	Budget
2023-02-01	Group 2	20000
2023-02-01	Group 3	7000
2023-03-01	Group 2	25000

Product

Product ID	Product Desc	Product Group
100	A	Group 1
110	B	Group 1
120	C	Group 2

Cardinality

Many to many (*:*)

Cross filter direction

Single (Product filters Budget)

☒ Make this relationship active
 ☐ Assume referential integrity
 ☐ Apply security filter in both directions

⚠

This relationship has cardinality Many-Many. This should only be used if it is expected that neither column (Product Group and Product Group) contains unique values, and that the significantly different behavior of Many-many relationships is understood.
 [Learn more](#)

OK

Cancel

Figure 7-5. Create a relationship between tables *Budget* and *Product*²

Use a bridge table

You can use a bridge table (see the “Budget” sections of Chapters 11, 15, and 19 for how to build such a table) and create two one-to-many relationships between the bridge table and the two tables (as shown in Figure 7-6). If Power BI should propagate a filter from the Product table to the Budget table, you need to set the relationship between the Product table and the bridge table to Both. This is one of the rare cases where the option of a bi-directional filter is good (you shouldn’t use it as your default filter direction). The two one-to-many relationships over the bridge table won’t have the disadvantages described for the many-to-many relationship. That’s why a bridge table is usually my preferred solution.

² A full-sized screenshot is available [online](#).

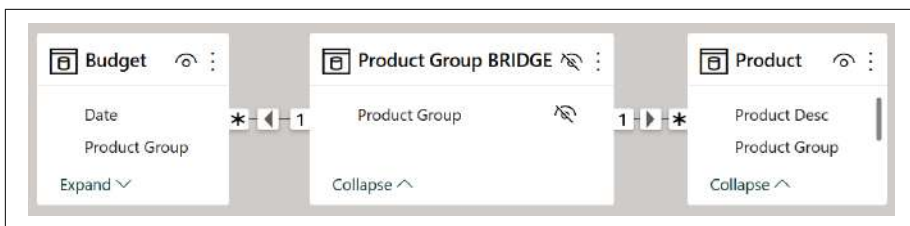


Figure 7-6. Using a bridge table between tables *Budget* and *Product*

In the next section, I demonstrate the solution I built for a customer that gives control over the display language to the report user.

Multi-Language Model

To build a data-driven multilingual data model as described in “[Multi-Language Model](#)” on page 63, you need to take care of the following things in Power BI:

- A dimension table for the available languages
- Visual elements (e.g., the report headline)
- Text-based content (e.g., product names)
- Numerical content (e.g., values in different currencies)
- Metadata (e.g., the names of tables and columns)
- The Power BI Desktop’s UI (both the standalone and Windows Store versions)
- The Power BI service UI
- The Power BI Report server UI

This section’s examples use [Multilanguage.pbix](#) from GitHub.

Dimension Table for the Available Languages

You need to create a table that contains a key and a display name for the available languages. I recommend setting the slicer/filter to single-select on all reports; it doesn’t make sense to select more than one language.

Table 7-1 has two languages: English and Klingon. You can use any value for column Language ID. I use the short names used by Azure Cognitive Services because I demonstrate its usage in “[Multi-Language Model](#)” on page 313 to automatically translate texts into different languages.

Table 7-1. A language table

Language ID	Language Desc
en	English
thl-Latin	Klingon

Visual Elements

You create a plain vanilla relationship between the Language table and the table containing the texts for the visual elements for all languages (see [Figure 7-7](#)).

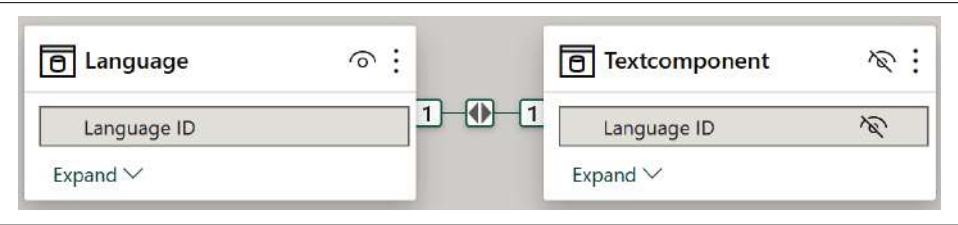


Figure 7-7. The Language table has a filter relationship with the Textcomponent table

I prefer having the table with the texts for the visual elements pivoted. That way, I have one column per visual element with one row per language. Because both the Textcomponent and Language tables have one row per language, the cardinality is one-to-one (see [Figure 7-9](#)). [Table 7-2](#) shows what this might look like.

Table 7-2. A (pivoted) table containing the texts for the visual elements

Language ID	SalesOverview	SalesDetails
en	Sales Overview	Sales Details
thl-Latin	QI'yaH	qeylIS belHa'

If you don't pivot this table, the cardinality of the relationship between tables Language and Textcomponent would be one-to-many instead, and you'd need to add a filter for the text box where you show the display name to filter on the right text identifier (see [Figure 7-8](#)).

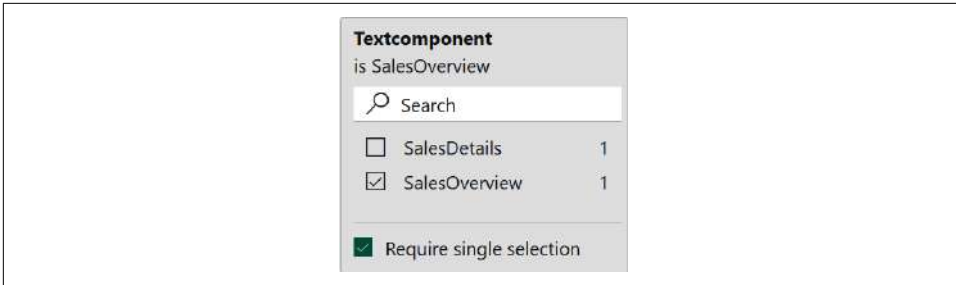


Figure 7-8. Filter on unpivoted table for visual elements

Text-Based Content

Now comes the complex part, which is the heart of my data-driven solution. In my solution, all dimension tables end up having a composite key composed of the dimension's key plus the language key. In order to identify a single row of a dimension table, it must be filtered on both the dimension key and the language key. The fact table references the dimension table only on the dimension key through a regular one-to-many relationship.

The language (key), on the other hand, is chosen by the report user via a slicer. To model this relationship is tricky. Of course, you can add a one-to-many relationship between the language table and the dimension table. But it turns out that this works only for the first dimension table. As soon as you add the relationship from the language table to the next dimension table, Power BI will complain about ambiguity. Adding another relationship of this kind would create more than one path from the language table to the fact table (one path is over the first dimension, the other path over the next dimension).

Power BI won't allow you to create an ambiguous model. It forces you to apply the tricks we learned when binning: leaving the dimension tables and the language table disconnected (as shown in [Figure 7-9](#)) and using DAX to create the relationship via function `TREATAS`, as you will learn in [“Multi-Language Model” on page 222](#).

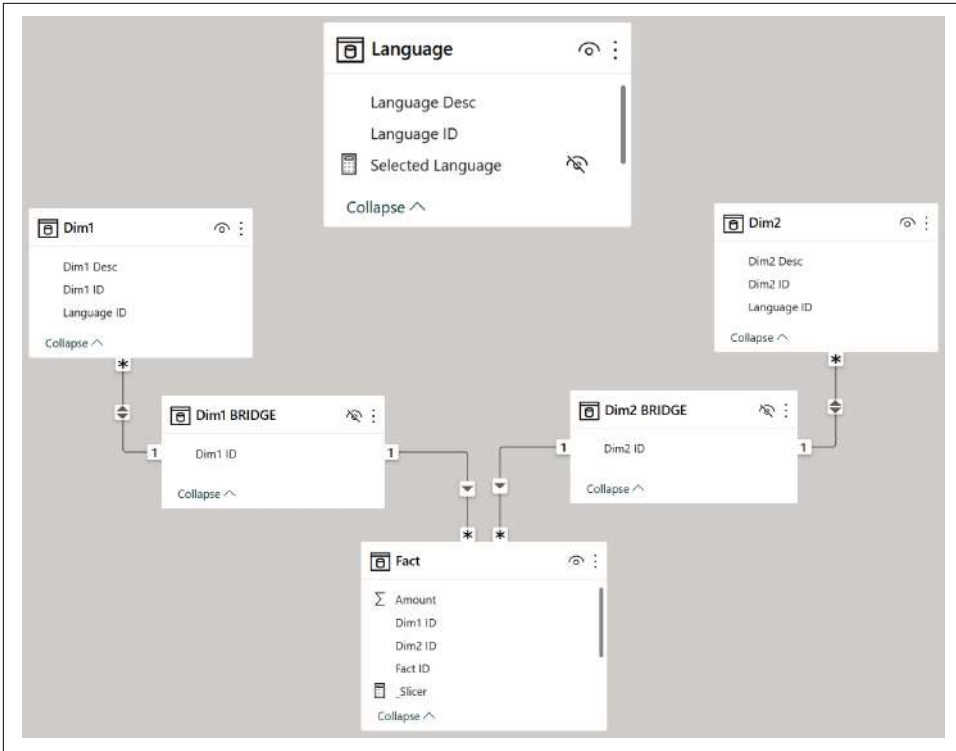


Figure 7-9. The Language table is disconnected from the dimension tables

Numerical Content

The table containing the exchange rates is usually disconnected (shown in [Figure 7-10](#)). Finding the correct row in this table can be accomplished with a non-equi-join (e.g., most current available exchange rate on or before the date the fact occurred). Again, we move the complexity over to DAX.

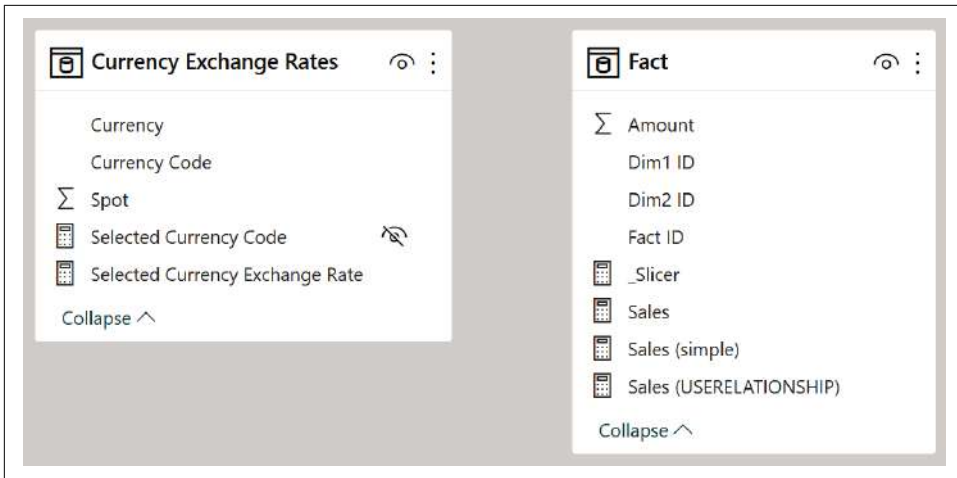


Figure 7-10. Tables *CurrencyExchangeRates* and *Fact* are disconnected

Data Model's Metadata

To add translations of the metadata, you need to export a JSON file, which will contain all artifacts of the data model. In this file, you add the translations and import it again.

Exporting the file for an Analysis Services tabular database is done via Visual Studio. At the time of this writing, Power BI Desktop has no feature to export the JSON file via the UI. The recommended tool to create the JSON file is *Tabular Editor*. It's not directly supported by Microsoft, but strongly recommended (you'll find references to Tabular Editor in all of Microsoft's official documentation, certification exams, etc.).



Tabular Editor was developed by Data Platform MVP Daniel Otykier and comes in two flavors; Version 2 is open source and free. For version 3 you need to buy a license. Find more about both versions at [Tabular Editor](#). Metadata export for Power BI (and Analysis Services tabular definition, for that matter) does work in both versions.

In *Tabular Editor's TOM Explorer* you will find *Translations* on the bottom of the list (see [Figure 7-11](#)). By right-clicking, you can export, and later reimport the JSON definition containing translations for all the data model's metadata.

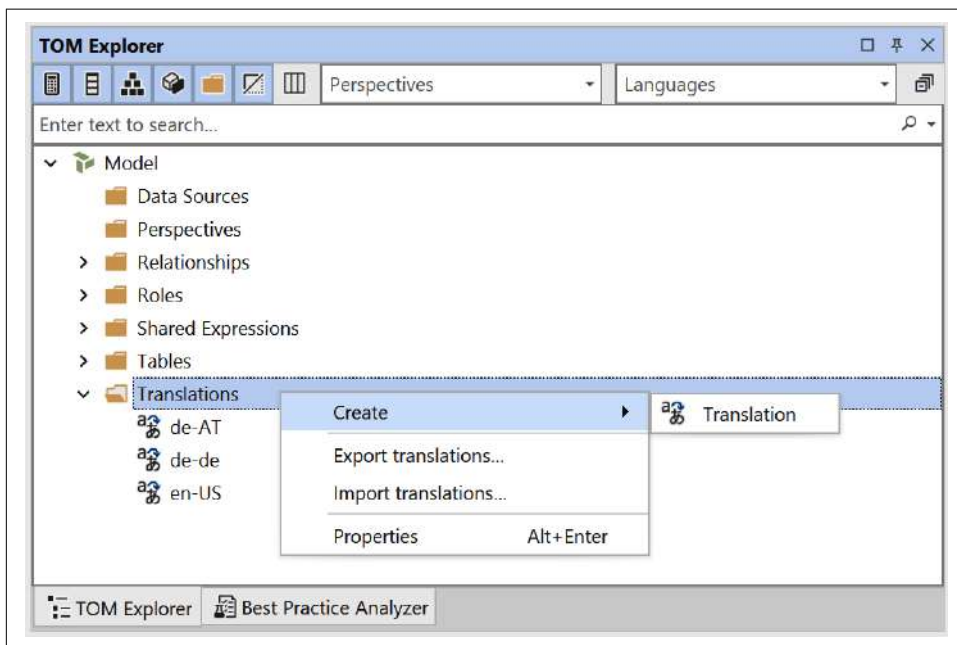


Figure 7-11. Tabular Editor's TOM Explorer lets you maintain translations

You can either edit the JSON file with a plain text editor or **Tabular Translator**, an open source and free tool by Kasper de Jonge (principal program manager at Microsoft).

In the JSON file, you can ignore the first half of the content and directly scroll down the section `cultures`. Tag `name` defines the culture you translate to. Afterward, you look for pair of lines containing `name` (the name the artifact has in the model) and `translatedCaption` (the name of the artifact in the new culture):

```
"cultures": [  
  {  
    "name": "de-AT",  
    "translations": {  
      "model": {  
        "name": "Model",  
        "translatedCaption": "Modell",  
        "tables": [  
          {  
            "name": "Language",  
            "translatedCaption": "Sprache",  
            "columns": [  
              {  
                "name": "Language ID",  
                "translatedCaption": "Sprache ID"  
              },  
              ... (cut off for brevity)
```

Tabular Translator makes entering the translations a bit more convenient, as you can see in [Figure 7-12](#). On top left, you see the culture. The first two columns of the grid in the middle show the table (e.g., `Language`, `Fact`, `Dim1`, etc.) and object type (e.g., `Model`, `Table`, `Column`, `Measure`, etc.). Then the original name, description, and display folder are displayed. Only the last three columns are editable. There you (or the translator) enter the translated versions of the name, description, and display folder. If you leave something empty, it will fall back to the original language of the model (English, in this case).

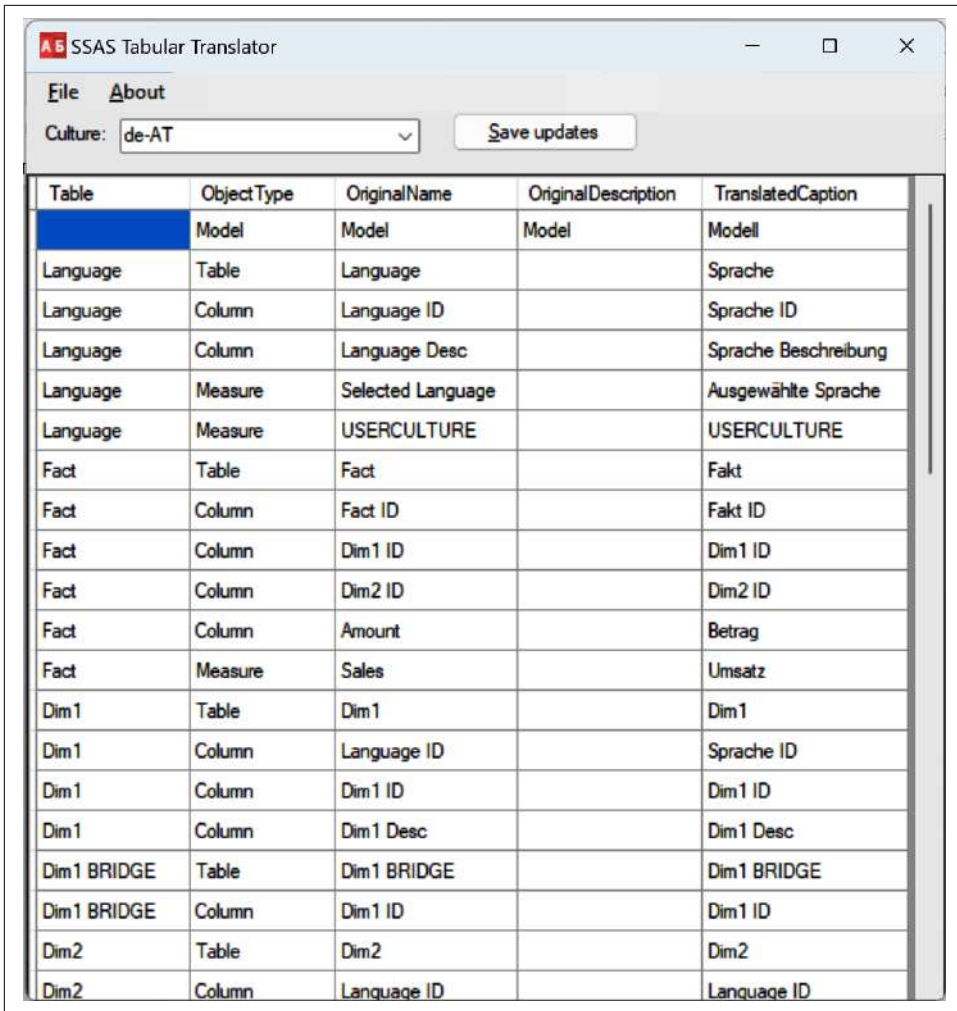


Figure 7-12. Tabular Translator is a convenient tool to maintain translations of metadata

UI of Power BI Desktop (Standalone)

Not matter in which language you initially installed Power BI Desktop, you can select from a long list of supported languages via File → Options → “Options and Settings” → Global → Regional Settings (see Figure 7-13). The “Application language” is language in which the Power BI Desktop displays the menu or messages and in which the data model (in the Data pane on the far right) is displayed. To activate this change, you need to close Power BI Desktop and start it again.

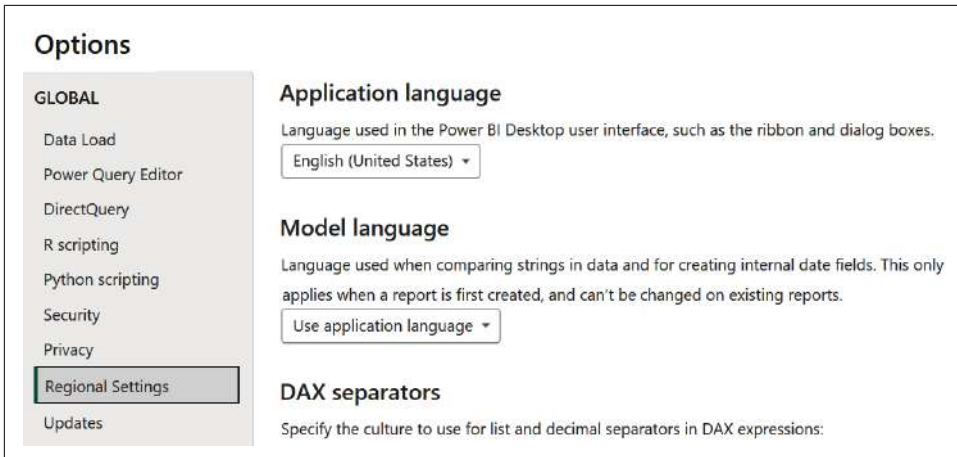


Figure 7-13. You can change the Application language and the Model language separately

UI of Power BI Desktop (Windows Store)

If Power BI Desktop isn't installed as a full application, but via the Windows store (which updates to the newest version automatically), then the setting from the operating system is used by the app and you can't change it separately. Look for "Time & language"—"Language & region" in the Windows settings if you need to change the display language (see Figure 7-14).



Figure 7-14. The Windows store version of Power BI Desktop respects the OS's display language

UI of the Power BI Service

Via the gear icon on the top right corner of the Power BI service, you will find General → Language (see [Figure 7-15](#)). There, you can change the display language by clicking “Select display language.”

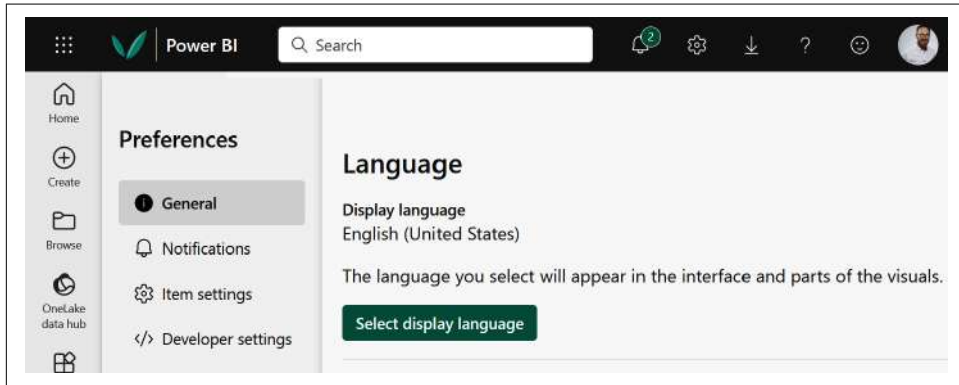


Figure 7-15. Selecting the display language in the Power BI service

UI of Power BI Report Server

The on-premises Power BI server respects the settings of your internet browser to choose the language of the user interface. In the case of Edge, you’ll find this setting via Settings → Languages → “Preferred language”). [Figure 7-16](#) shows the setting in the Edge browser.

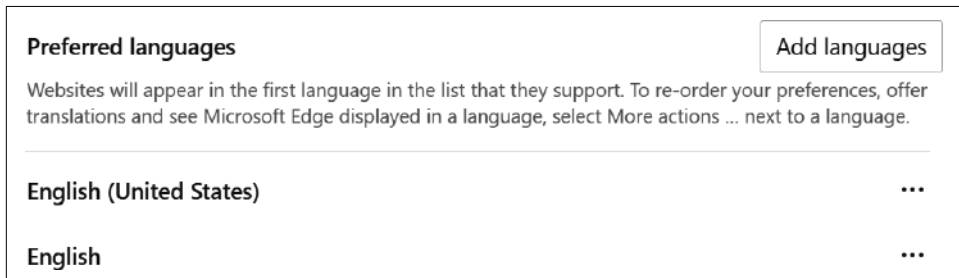


Figure 7-16. Selecting the display language in Edge

Next I show an interesting way to model key-value pair models to reports so that the the report-user has a maximum flexibility to analyze correlations in the data.

Key-Value Pair Tables

In the Model view of Power BI, a key-value pair table is just a single table without any special relationships. To transform it into a proper model, you need to pivot the table (which you can learn in the “Key-Value Pair Tables” sections in Chapters 11, 15, and 19) and then split it into fact tables and dimension tables, as discussed in Chapter 6.

But there’s one interesting use case for a very flexible kind of report, where you not only keep the key-value pair table in its unpivoted state but load it twice into Power BI. Figure 7-17 shows the key-value table Source twice in Power BI’s data model.³ The filter relationship is created on the ID column. The cardinality is many-to-many (as the same ID can appear multiple times in the key-value pair table) and the filter direction is single-directed from Source 2 to Source.

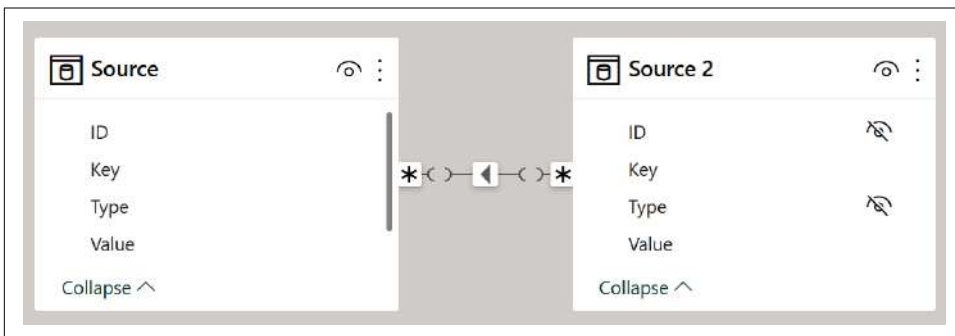


Figure 7-17. The key-value pair table loaded twice into the data model

Modeling the key-value pair this way enables you to create a report like Figure 7-18. The two slicer visuals filter the Key column of table Source once and table Source 2 once. The matrix in the center shows the Value column of table Source in the rows and of table Source 2 in the columns. In the Values section of the matrix visual, I put a measure counting the rows of the Source table:

```
[Count of rows] := COUNTROWS('Source')
```

³ The examples in this section use the *KeyValue.pbix* file from the book’s GitHub repository.

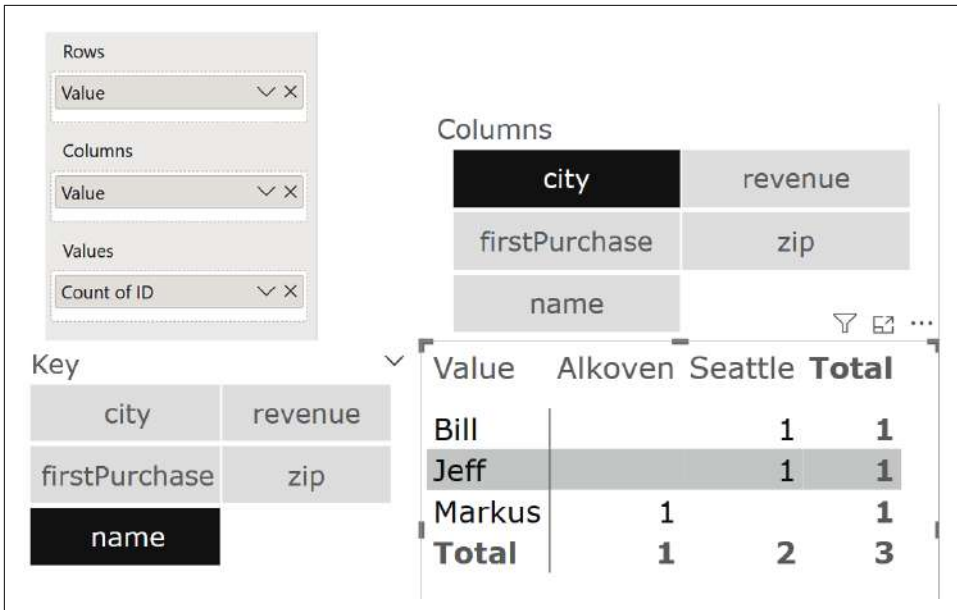


Figure 7-18. A flexible report build on top of the duplicated key-value pair table

Now the report user has full flexibility on what (key) to display on the rows and on the columns of the matrix visual. As a result, you see how many entries you can find in the key-value pair table for the different combinations of values. For example, there is one Bill, one Jeff, and one Markus in the table. Bill and Jeff live both in Seattle, while Markus lives in Alkoven. In total, we're talking about three different IDs in the key-value pair table.

Independently of the shape of your data model, you'll face situations where you want to enrich an existing data model with local data or combine two models with each other. This problem is usually described as the combination of self-service and enterprise business intelligence. Power BI comes with an very interesting approach, as you will learn next.

Combining Self-Service and Enterprise BI

Power BI Desktop was born as a self-service BI tool in 2015 and comes with plenty of features to make the lives of information workers easy. Most importantly, you can import data from a wide variety of data sources into Power BI Desktop. But it also allows you to connect to a relational data warehouse or an analytic database (e.g., hosted on an Analysis Services tabular, or a Power BI semantic model, hosted in the Power BI service). These data sources are usually created, or at least curated, by IT departments.

The catch with a connection to a Power BI semantic model or Analysis Services, though, is that this is considered a *live connection*. And a live connection comes with limitations. For example, you can only create a connection to one single data source; that is, only one Power BI semantic model or one single Analysis Services database. It doesn't allow you to add data from any other data source (e.g., an Excel spreadsheet, relational data warehouse, etc.). As you can see in [Figure 7-19](#), you can create a *New measure* but not add new columns.

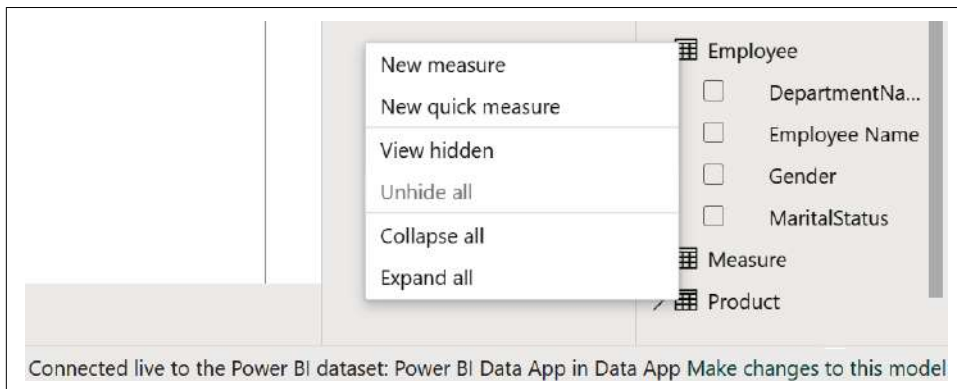


Figure 7-19. Limited functionalities in live connection mode

Since April 2023, an alternative has generally been available: a DirectQuery connection to a Power BI semantic model or Analysis Services tabular (not multi-dimensional, though). Don't misinterpret live connection and DirectQuery as being similar. Despite the names, under the hood these are very different technologies. (In both, the data is only queried as needed by the report consumer and not loaded into Power BI Desktop).

The most important difference is that you can create a data model that mixes the two storage modes Import and DirectQuery. You can load the data for some of the tables but keep other tables in DirectQuery mode. This is called a *composite model* (two storage modes are combined in the model). Both the imported tables and the DirectQuery tables can come from different data sources. (A live connection cannot be part of a composite model.)

This option allows you to create a model that reuses all the centralized and predefined logic of your enterprise model but adds information (e.g., an Excel spreadsheet containing the first draft of your budget or a CSV file containing a custom definition of sales regions or data from a different enterprise data model) into the same model. Based on this (composite) model, you can then create a report containing both self-service and enterprise information.

The only thing you have to keep in mind is finding the right relationships between tables from the different data sources (because, for example, the products might have different key values in the data warehouse than they have in the Excel spreadsheet). If you add fact tables from different sources, then you will face the problems of multi-fact models. You can then apply the solutions discussed in “[Budget](#)” on page 135.

Key Takeaways

In this chapter, you took a deep dive into advanced data modeling concepts. Basically, all of these advanced challenges can be solved only with Power BI because it is a model-driven tool. I don’t want to imagine solving these problems in one big table (in Excel) or by writing the SQL code for a fully normalized data model in my reporting tool. I hope that these challenges and solutions make clear why having a model-driven tool is an advantage, even if it might have looked overly complex at first. Here is what you’ve learned:

- A lookup table for the bins must be connected as a regular one-to-many relationship to the fact table.
- To connect a fact table that’s of a coarser granularity than the primary key of the dimension table, you have three options: use DAX, create a many-to-many relationship, or introduce a bridge table to the model.
- Setting the language for the text in the report, the names in the data model, and the language of the application UI you’re working with is relatively straightforward.
- The lookup tables for the language must stay disconnected from the dimension tables because Power BI doesn’t allow ambiguous data models.
- The lookup table for the currency exchange rates stays disconnected for a different reason: finding the correct date of the exchange rate involves a non-equi-join (which only can be solved with DAX).
- Power BI allows you to create composite models where tables can have independent data sources and can be in different storage modes (Import or DirectQuery). This bridges the gap between pure self-service and pure enterprise BI solutions.

The next chapter elaborates on the different storage modes: Import, live connection, and DirectQuery.

Performance Tuning in the Power BI Data Model

Performance tuning in Power BI is a complex topic with several components involved. Every storage mode has different performance implications, and optimizing the report runtime must be done with approaches specific to the storage mode. You have more than one option for implementing the same calculation in DAX. You can pre-calculate values and physically store them in the data model, or first let calculations be done ad hoc via explicit measures. The in-memory compression is highly dependent on the cardinality of a column—and, therefore, can't be predicted in a general way.

Performance tuning in Power BI could easily fill a book on its own. All the optimizations have one thing in common, though: scanning less data will speed up query time. Because this book is about data modeling, I limit this chapter to the storage modes and how the variants can be combined in a data model to speed up query time. You can use the *Performance Tuning.pbix* file to follow along with the examples in this chapter.

Storage Mode

The VertiPaq engine offers different storage modes. Some of them can be set under the Advanced settings in the Properties pane for a table (Figure 8-1).

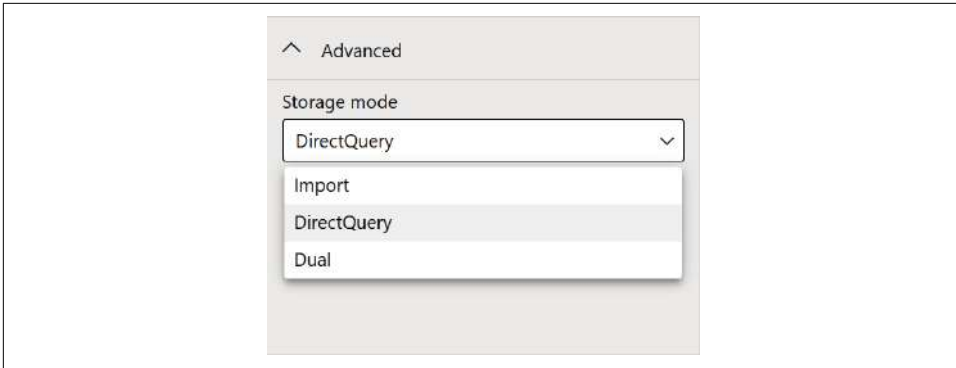


Figure 8-1. Storage mode of tables

The storage modes in VertiPaq engine include the following:

Import mode

When you import data, the complete table is physically loaded into the data model. The data gets compressed and stored in an in-memory columnar database. This offers the best query performance by far. Therefore, this should be the mode of choice in many scenarios. This mode works for all data sources (from flat files over Excel worksheets to all kinds of databases). A regular refresh needs to be scheduled to keep the stored data up-to-date.

DirectQuery mode

In this mode, only data connection information (like server name and database name) and metadata (like names of tables and columns) get stored in the data model. Every query against the VertiPaq engine (like a DAX statement generated by a visual inside Power BI) is sent to the data source instead of any data being stored locally. If necessary, the DAX query will automatically be converted into a SQL query.

This mode is available only for some data sources (namely relational databases, which can be queried in SQL, and analytical databases like Power BI semantic models hosted in the Power BI service or Analysis Services tabular, which can be queried in DAX).

Performance, especially for relational databases, is slower compared to Import mode. There's some overhead involved when running a query that combines information from different data sources. The SQL statement (derived from the DAX query) is less than optimal, and the performance of the query is fully dependent on the data source. It's a challenge for any data source to compete with the query performance of the VertiPaq engine, even when tuned properly. If you don't have the knowledge or opportunity to invest in performance tuning of

the data source, then DirectQuery mode isn't for you. Some limitations apply to DirectQuery mode, which [Microsoft documents](#).

No data refresh needs to be scheduled in this mode, as no data is stored inside of Power BI. When your data model is in DirectQuery mode, you see a hint in the status bar on the bottom of the window, as shown in [Figure 8-2](#).



Figure 8-2. DirectQuery mode

Dual mode

A table in Dual mode covers both features: it's imported into the data model and all connection metadata are kept in the data model, so it can be used in DirectQuery mode as well. In [“Dual Mode” on page 169](#), I explain in which scenarios it makes sense to set a table to this storage mode.

Live connection mode

Live connection is similar to DirectQuery in the sense that no data is loaded into the VertiPaq engine. But it is completely different if you look at the technology under the hood. You can't set storage mode to live connection on a table-by-table basis (as you can with the other storage modes, see [Figure 8-1](#)). Live connection mode is possible only for analytical databases, which can be queried in DAX (as a Power BI semantic model, in Azure Analysis Services, or in SQL Server Analysis Services tabular). The whole data model is then in live connection mode (unless you choose to import all necessary tables or choose DirectQuery mode when you make the first connection to the data source database).

In live connection mode you can create explicit measures, but many other data modeling features are not available (like using the Power Query editor to transform the data or add calculated columns or calculated tables). If you need such features, switch to DirectQuery mode instead.

Live connection is the default mode when connecting to a Power BI semantic model or Analysis Services tabular databases. You can convert such a live connection with just a few clicks into a DirectQuery connection, by clicking on “Make changes to this model” on the bottom of the screen, as shown in [Figure 8-3](#).



Figure 8-3. Live connection mode

Power BI asks you for confirmation before it executes the conversion ([Figure 8-4](#)). You do this by clicking “Add local model.”

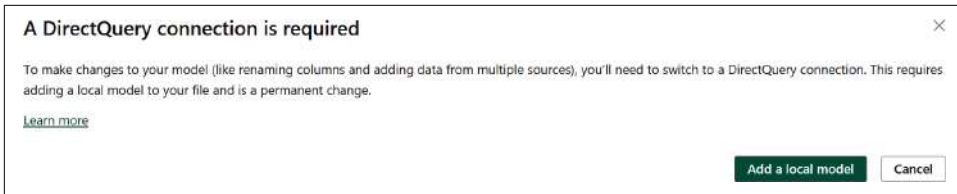


Figure 8-4. Converting to a DirectQuery connection¹

In the next step, you need to decide which parts of the remote data model should be made available to the current data model (Figure 8-5). Under Settings, you can decide if Power BI should add tables that are added later automatically to the model (which is the default), or not.

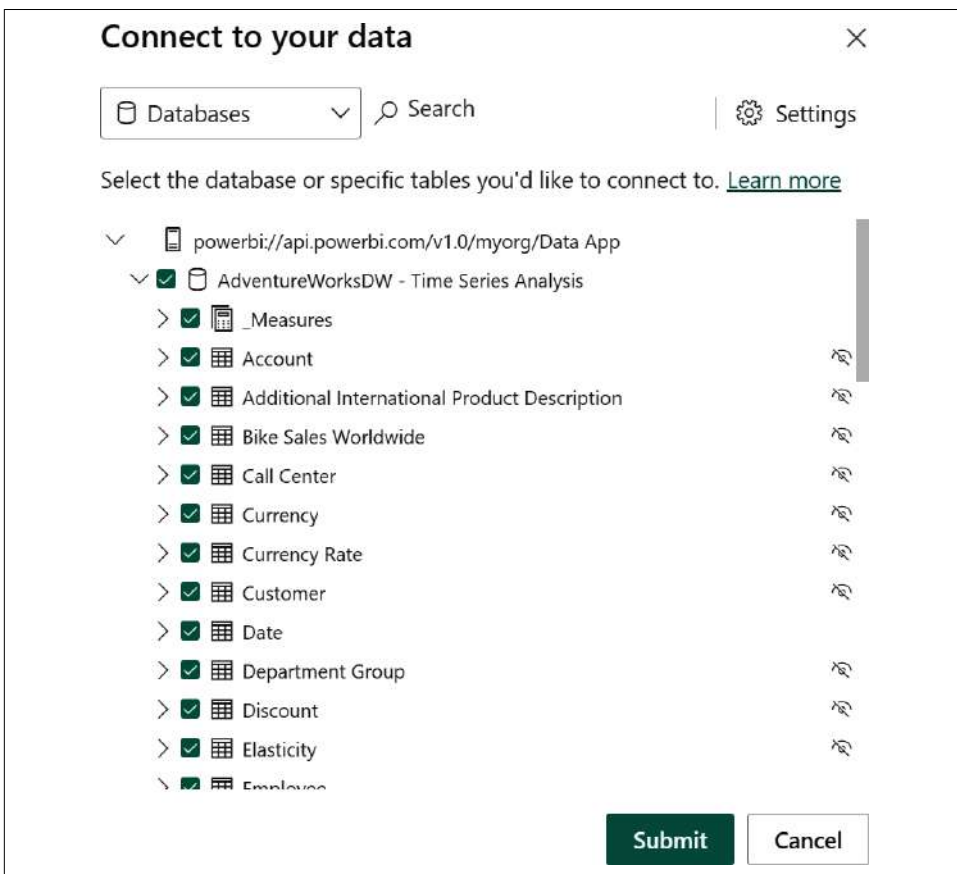


Figure 8-5. Choosing the tables

¹ A full-sized screenshot is available [online](#).

Click Submit to actually convert the data model and load its metadata (see [Figure 8-6](#)).



Figure 8-6. Loading the (meta) data

The status bar on the bottom of the window will change to “Storage mode: DirectQuery,” as you saw in [Figure 8-2](#)).



Converting from live connection to DirectQuery cannot be undone. Therefore, I recommend making a backup of the PBIX file before you convert the type of connection, just in case.

While your data model is in DirectQuery mode, you can still decide to import additional tables into the Power BI data model or add another DirectQuery data source. If you do so, Power BI will warn you about a *Potential security risk* ([Figure 8-7](#)).

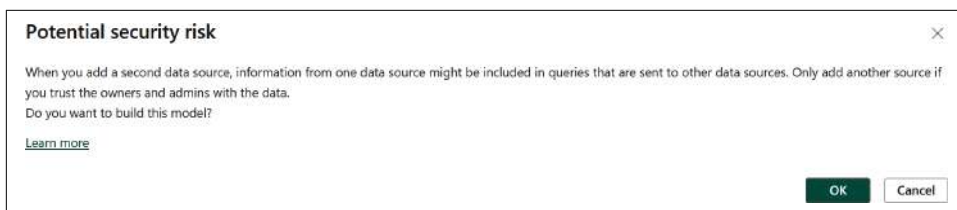


Figure 8-7. Warning about a potential security risk²

This *potential* security risk becomes a *real* security risk under two circumstances:

- You or any consumer of the data model combines information from both data sources in a single visual. In that case, Power BI doesn’t even temporarily load the data from both DirectQuery data sources to combine it into a single query result. Instead, it pushes the join key’s values as hardcoded values into the query sent to one of both DirectQuery data sources.

² A full-sized screenshot is available [online](#).

- The join key for both tables contains sensitive information that must not be potentially seen by anybody with access to the query code. An administrator of a database, for example, has access to the queries running against the database she is responsible for. In a man-in-the-middle attack, someone would be able to capture the code of the queries sent to a database.

In the following example, you'll see the portion of a SQL query generated by Power BI in order to join information from a date table in "Import mode" (with alias `semijoin1`) with a fact table in DirectQuery mode (with alias `basetable0`). The whole list of values of the date table's date column becomes part of the query sent to the DirectQuery data source. This code can be seen by the database administrators or attackers who can capture the query code. I don't consider the list of dates as sensitive, so I don't see a problem for this concrete query:

```
...
INNER JOIN
(
  (SELECT 368 AS [c42],CAST('20230101 00:00:00' AS datetime) AS [c16]) UNION ALL
  (SELECT 369 AS [c42],CAST('20230102 00:00:00' AS datetime) AS [c16]) UNION ALL
  (SELECT 370 AS [c42],CAST('20230103 00:00:00' AS datetime) AS [c16]) UNION ALL
  ...
  (SELECT 730 AS [c42],CAST('20231229 00:00:00' AS datetime) AS [c16]) UNION ALL
  (SELECT 731 AS [c42],CAST('20231230 00:00:00' AS datetime) AS [c16]) UNION ALL
  (SELECT 732 AS [c42],CAST('20231231 00:00:00' AS datetime) AS [c16])
)
  AS [semijoin1] on
(
  ([semijoin1].[c16] = [basetable0].[c16])
  ...

```

As soon as you combine tables with different storage modes in one data model, we speak of a composite model, and the status bar shows Storage Mode: Mixed (see [Figure 8-8](#)).



Figure 8-8. A composite model has a mixed storage mode

It's common to speak of *islands* or *source groups* when someone refers to data in different storage modes and/or from different DirectQuery data sources. In this sense, imported data always belongs to the same island, no matter which data source it was imported from. Data in DirectQuery mode belongs to the same island only if the tables are from the same data source.

This concept is important when it comes to performance: the potential security risk for cross-island queries is also a potential performance risk. Executing queries with long code (due to the injection of filters as hardcoded filter values into the query

code) will put more pressure on the source system, similar to when a filter is applied as an inner join between two physical (and indexed) tables inside one database.



All relationships across different islands, independent of their cardinality (like one-to-many, one-to-one, or many-to-many), are *limited relationships*. You learned about this in “[Relationships](#)” on [page 88](#) and can find more in [Microsoft’s documentation](#)). In the Model view, such relationships are represented with parenthesis-like marks after the cardinality indicators. Tables are joined with an *inner join* (as opposed to an *outer join* for regular relationships), and no blank rows are shown for rows violating the referential integrity.

Maybe you’re wondering when to choose which storage mode. I suggest going through the following list before you make your decision:

- When your data source is a Power BI semantic model or an Analysis Services database, choose live connection. Performance for reports will be very good.
- When you need to enrich the data model provided as a Power BI semantic model or an Analysis Services database, migrate to DirectQuery mode and import the additional information. Pay special attention to the performance of cross-island queries.
- When your data source is not a Power BI semantic model, you should consider importing all the data into the data model. It will give you a superior report performance.
- After at least one of the following criteria is met, evaluate if you can live with all the limitations of DirectQuery and a less-than-optimal query performance:
 - Use DirectQuery when the amount of data is too big to load into Power BI. “Too big” in this case means it exceeds either the physical limitations or those of your budget. Keep in mind that Power BI Premium and Microsoft Fabric always store data in a compressed format and that a dataset can contain up to 100 TB. 100 TB of compressed data can easily hold uncompressed information of beyond 1 petabyte (= 1,000 TB).
 - Use DirectQuery when the data source has complex data source has complex **row-level security in place**, which can’t be replicated in Power BI or Analysis Services tabular. Keep in mind that row-level security can either be implemented over (Azure Active Directory) roles or in a dynamic fashion inside the data model.
 - Another reason for using DirectQuery could be that refreshing the data model takes too long. It’s “too long” if the refresh time puts too much pressure on the data source or renders the data stale again by the time the refresh finally

finishes. In a (near-)real-time scenario, the latter can even be a very short period of time (like a couple minutes or seconds). Keep in mind that with incremental refresh or your own partition strategy, you can speed up data refresh so that the imported data stays fresh (see “[Partitioning](#)” on page 160). Make sure to compare the query-runtime between a model over the same data in Import mode and in DirectQuery mode to ensure that the DirectQuery mode doesn’t sabotage your real-time goals with too-long runtimes (which can be minutes or even run into a timeout).

Performance Analyzer

Chose View → Performance Analyzer to collect information about the runtime of visuals built in Power BI Desktop. These collections will tell you how much time was spent on running the DAX query (executed locally) and/or direct query (executed on a remote database), how long it took Power BI Desktop to “draw” the visuals (“Visual display”) or how much time a visual waited for Other visuals to finish.

Via “Copy query,” you get access to the actual DAX (or SQL) queries executed. Paste this to either [DAX Studio](#), an open source tool by Darren Gosbell (senior program manager at Microsoft), or [SQL Server Management Studio](#) to further investigate the query plans or optimizations.

Optimization of DAX queries and SQL queries is beyond the scope of this book.

Independently from the available storage modes, you can decide to partition a table into easier-to-manage subentities.

Partitioning

Partitioning is a way to split a table into smaller parts (partitions). Instead of one big storage entity covering all rows, you end up with several smaller storage entities with fewer rows each. This can improve query time, because scanning for data can be limited to the partitions whose metadata indicate that they might contain necessary information. But more importantly, you can schedule a refresh on the partition level.

Instead of always triggering a full refresh (for the full content of a table), you can set the trigger to only update certain partitions; for example, those containing data from the last few days, where a change in the data source could have happened. That said, it makes sense to partition your big tables by a timestamp related to a transaction.



At the time of writing, creating partitions is only available through the XMLA endpoint, which is a premium feature in Power BI/Fabric.

Choosing the Partition Key

Don't partition data on metadata like creation date or modification date. First, such a timestamp probably won't be used for filtering in most of your reports. Only a minority of users are interested in when a fact is created; more people are interested in the day a fact is related (when the fact takes place). A fact might be created in the past or future. For the best performance, the partition key must be the first column of every index (so that the index is aligned with the partitions). If the partition key isn't part of the filter, then the index won't be used.

Secondly, the modification date of a row can be changed (with every modification) and will then move a row between partitions. If the old and the new partition are both not refreshed, you end up with duplicates of this row in your data model (as the row will load into the new partition but still be in the old partition as well, as long as the old partition is not refreshed).

If you don't know what the previous value of the modification date was, you need to update *all* partitions. This would defeat the purpose of creating partitions.



Use a fact-related timestamp instead: the booking date or the day of order, which will not change at a later point in time.

In one project, I was confronted with the requirement that a change in the data source be available in a Power BI report within 10 minutes. It turned out that even with heavy performance tuning in the data source, these complex queries would take longer than 10 minutes on average in a DirectQuery setup to finish. (So much for “real-time” and DirectQuery.)

To solve it, I partitioned the table. As a partition key, I used a special column created for the sole purpose of partitioning. Every time the source table changed, the row's partition key was logged in a separate table. Every five minutes, a job was triggered to read this separate logging table and trigger a data refresh for those partitions where changes had happened since the last run of this job. After we found the right number of partitions (i.e., not containing too many rows so the refresh is fast enough, but also not having so many partitions as to refresh all the time), the regular refreshes took

only a couple of minutes. And the report response time with the imported data was under a second (compared to over 10 minutes with the DirectQuery approach).

Power BI Desktop allows you to define partitions only over the feature *incremental refresh*. As the name suggests, it enables you to refresh the data of a table in increments. You need to follow three steps if you want to enable this feature for a table. First, you need to define two Power Query parameters with the mandatory name *RangeStart* and *RangeEnd*. In Power Query, select Home → Manage Parameters and create them via *New*. As you can see in [Figure 8-9](#), these parameters must be of type Date/Time.

The screenshot shows the 'Manage Parameters' dialog box. On the left, under the 'New' tab, there is a list of parameters: ServerName, DatabaseName, SchemaName, TableName, RangeStart (selected), and RangeEnd. Each parameter has a small icon next to it. On the right, the configuration for the selected 'RangeStart' parameter is shown. It includes a 'Name' field with 'RangeStart', a 'Description' field, a 'Required' checkbox which is checked, a 'Type' dropdown set to 'Date/Time', a 'Suggested Values' dropdown set to 'Any value', and a 'Current Value' field with '01/01/2019 00:00:00'. At the bottom right, there are 'OK' and 'Cancel' buttons.

Figure 8-9. Power Query parameter for the start of the time range of an incremental refresh

Second, you need to use both parameters in the Power Query of the fact table (for which you want to turn incremental refresh on) as a filter. [Figure 8-10](#) shows how you create a *Between* filter on the *OrderDate* column, which is the partition key in this example.

The filter itself references the two parameters, making sure that only rows where the *OrderDate* is *after or equal to* *RangeStart* and *is before* *RangeEnd* are loaded from the data source, as shown in [Figure 8-11](#).

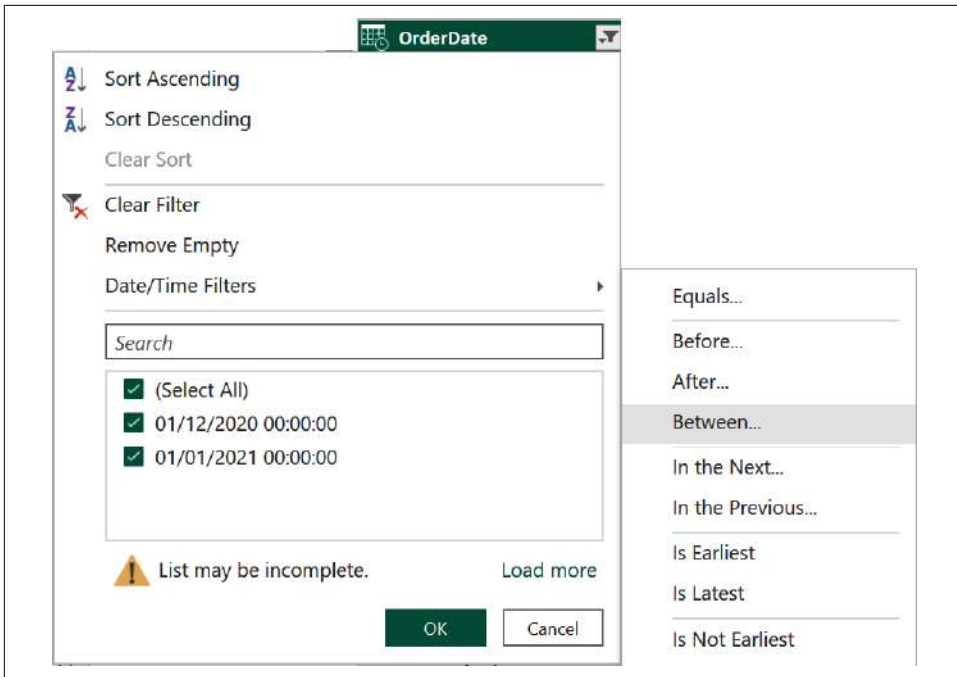


Figure 8-10. Creating a filter on the partition key

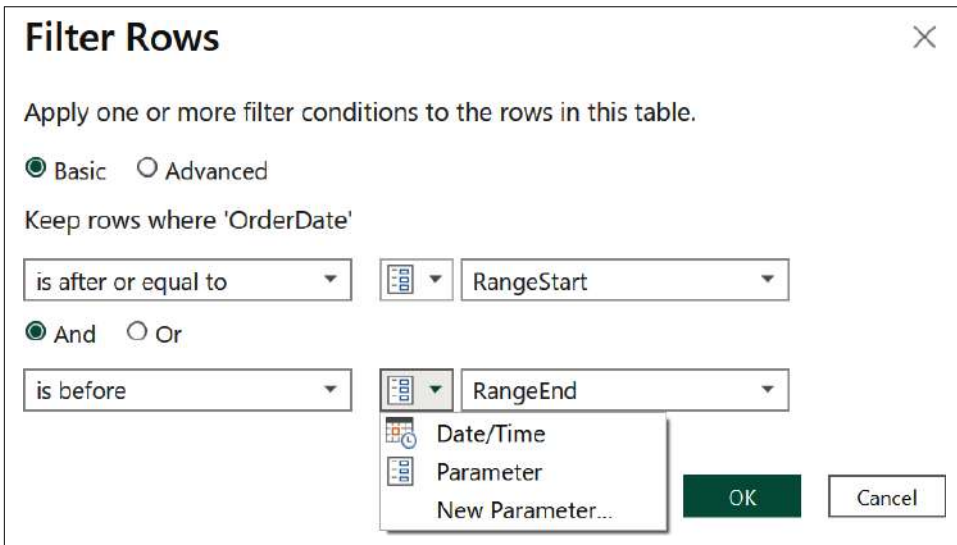


Figure 8-11. Filtering the partition key with the parameters

In the Model view, you need then to right-click the table and choose incremental refresh. Next, take these four steps (as shown in [Figure 8-12](#)):

1. Select table

The table is already pre-selected, but you can choose any table of the data model.

2. Set import and refresh ranges

Turn on “Incrementally refresh this table.” This is only possible if you’ve already defined the `RangeStart` and `RangeEnd` parameters in Power Query. You then decide how many periods of data you want to keep in the table. I discard anything older than 5 years in this example. Additionally, I set “Incrementally refresh data starting” to 1 month before refresh date.

3. Chose optional parameters

I keep “Get the latest data in real time with DirectQuery (Premium only)” disabled for this example. In [“Hybrid Tables” on page 170](#), you can learn about this option. I enable “Only refresh complete month.” If you choose to “Incrementally refresh data starting” in days or years, this setting will offer accordingly complete days or years as well. If you enable “Detect data changes,” a refresh will not run by a fixed cadence, but only when data changes. Refer to [the Microsoft documentation](#) to learn more about this feature.

4. Review and apply

Here you see a graphical representation of the settings you enabled.

Partitioning splits existing data into smaller parts. In the next section, you’ll learn how you can optimize query runtime by intentionally introducing duplication of data in an pre-aggregated form.



At the time of writing, enabling incremental refresh will disable the option to [download a Power BI file from the Power BI service](#).

Incremental refresh and real-time data

Refresh large tables faster with incremental refresh. Plus, get the latest data in real time with DirectQuery (Premium only). [Learn more](#)

i These settings will apply when you publish the dataset to the Power BI service. Once you do that, you won't be able to download it back to Power BI Desktop. [Learn more](#)

1. Select table

Reseller Sales (Incremental Load) ▾

2. Set import and refresh ranges

☒ Incrementally refresh this table

Archive data starting

Years ▾

 before refresh date

Data imported from 1/1/2018 to 4/30/2023 (inclusive)

Incrementally refresh data

Months ▾

 before refresh date

starting

Data will be incrementally refreshed from 5/1/2023 to 5/31/2023 (inclusive)

3. Choose optional settings

☐ Get the latest data in real time with DirectQuery (Premium only) [Learn more](#)

☒ Only refresh complete month [Learn more](#)

☐ Detect data changes [Learn more](#)

4. Review and apply

Archived

Incremental Refresh

5 years before refresh date

1 month before refresh date

Refresh date

Apply

Cancel

Figure 8-12. The dialog box guides you through the steps necessary to turn incremental refresh on for a table

Pre-Aggregating

There's no reason you couldn't load the same piece of information several times into a data model. For example, you can load a transaction table on the finest necessary granularity (so that even very detailed analytic requests are satisfied) and load it once again on an aggregated level (e.g., one row per day and product). This enables your data model to satisfy detailed information (from the transaction table) and do fast calculations (from the aggregation table) as well.

This concept works for both imported data or tables in DirectQuery mode. Detailed data may be needed only very rarely. If this is the case, then you could keep the transaction table in DirectQuery mode to save storage and refresh time and import only the aggregated version of this table to enable fast reports for the majority of analysis. Reports on the detailed data will take longer to execute, but that's sometimes acceptable.

For rather simple requirements, Power BI does fully support such a performance optimization. You can specify the granularity level for which the aggregation table aggregates data, which is then available in the detailed table. The detailed table must be in DirectQuery storage mode. In Chapters 12, 16, and 20, you'll learn how to create a aggregation table. Here I show you how to tell Power BI in which cases it should use the aggregation table instead of the detailed table to satisfy queries.

Figure 8-13 has two tables: Reseller Sales (DirectQuery - Agg), which contains all rows in the most detailed granularity (date and product), and Reseller Sales (Agg Table PQ), which contains the same information, but aggregated only by date (with the product key omitted).

There is no filter relationship between these two tables. If you right-click Reseller Sales (Agg Table PQ) and choose "Manage aggregations" (**Figure 8-13**), the selected table is already pre-selected (but you could chose a different one via the list box "Aggregation table"). If more than one aggregation table could satisfy a query, then the Precedence is used to decide which one to choose. The aggregation table with the higher value in Precedence is preferred.

The aggregation table in **Figure 8-13** contains three columns. For each, you need to tell Power BI how it relates to the detailed table or the rest of the model. In my example, the column OrderDate is used to GroupBy column Date of table Date (Direct Query). Column SalesAmount is the Sum of my detail table's (Reseller Sales (DirectQuery - Agg)) column SalesAmount. And column SalesCount calculates a "Count table rows" again over the detail table (Reseller Sales (DirectQuery - Agg)). Based on this information, Power BI makes informed decisions when an aggregation on column SalesAmount of the detail table (Reseller Sales (Direct

Query - Agg)) can be satisfied with an aggregation on the aggregation table (Reseller Sales (Agg Table PQ)) instead.

Manage aggregations

Aggregations accelerate query performance to unlock big-data sets. [Learn more](#)

Aggregation table

Precedence ①

Reseller Sales (Agg Table PQ)

0

AGGREGATION COLUMN	SUMMARIZATION	DETAIL TABLE	DETAIL COLUMN	
OrderDate	GroupBy	Date (DirectQuery)	Date	
SalesAmount	Sum	Reseller Sales (Direct...	SalesAmount	
SalesCount	Count table rows	Reseller Sales (Direct...		

Apply all

Cancel

Figure 8-13. Managing aggregations in your data model

To prove that Power BI uses the aggregation table correctly, I created a report with two table visuals: both contain the same measure, Sales Amount (DirectQuery → Agg) (which is defined as the sum of column SalesAmount of the detail table, SUM('Reseller Sales (DirectQuery → Agg)'[Sales Amount])) and the CalendarYear column (from table Date (DirectQuery)). One additionally contains column Style from the Product table. Because the aggregation table doesn't contain product specific information, but is aggregated only on the day, Power BI is expected to use the aggregation table to calculate the measure in only the first visual. Only the detailed table contains product-specific information; therefore, the second visual is expected to use the detailed table.

In the Performance Analyzer pane on the of [Figure 8-14](#), you can see that for the first visual (named "Date only"), only a DAX query is available. Obviously, the aggregation table was used, which is in Import mode. DirectQuery is mentioned in the second visual (named Date & Product). That's the sign that the detailed table (which is in DirectQuery mode) was used to satisfy the query.

In more advanced scenarios (e.g., when the detailed table is in Import mode), you can't use Power BI's "Manage aggregations." You need to add logic to your measures instead. The code in the measure decides from which table it can satisfy a calculation in order to achieve the best possible performance. [Chapter 12](#) explains how you can create such special measures in DAX.

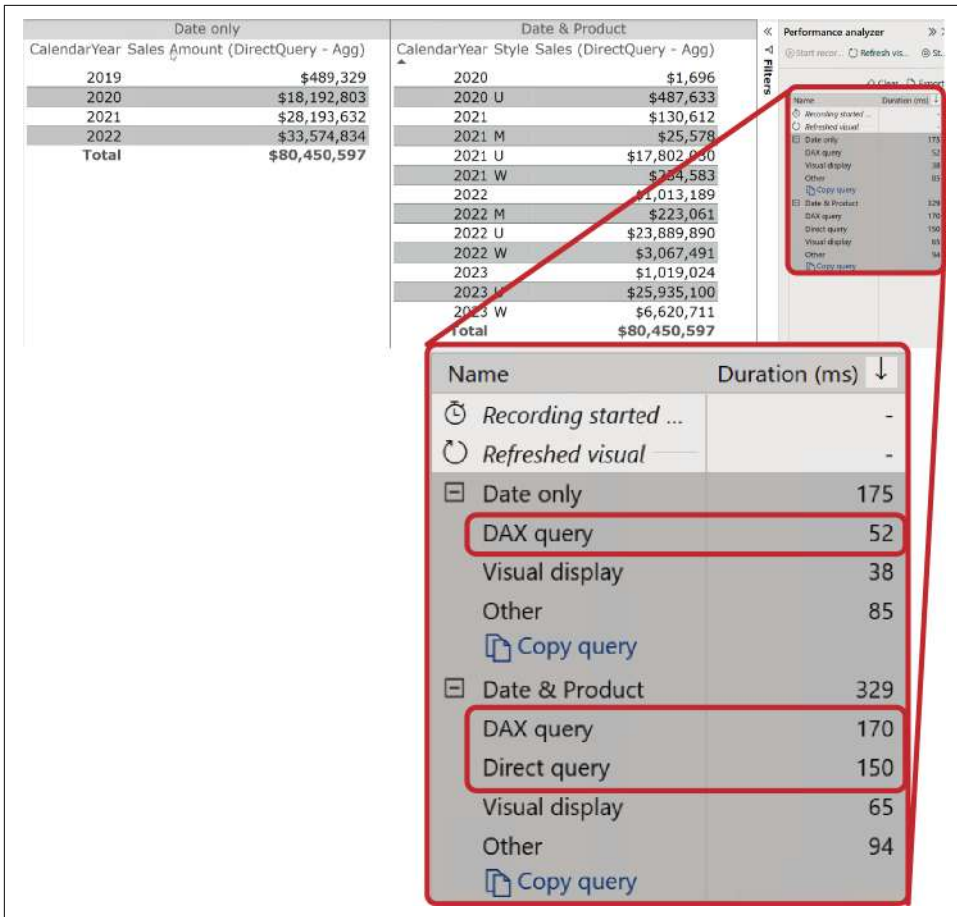


Figure 8-14. Analyzing the performance of reports of different granularity

Composite Models

Recall from “Storage Mode” on page 153 that Power BI allows you to set the storage mode per table. You can build a data model where some of the tables are imported, while others are in DirectQuery mode. (Composite models aren’t possible in live connection mode.)

This feature helps to design performant data models. For example, you can import all the (small) dimension tables into the data model but keep the big dimension tables and the fact tables in DirectQuery mode. You can also create aggregation tables (see “Pre-Aggregating” on page 166) for those big tables, which you either import (to give queries on the granularity level of the aggregation table the best possible performance) or keep in DirectQuery mode (to save storage space and refresh time and still

give a better performance, if the query to get the aggregated data is properly tuned for performance on the data source's side).

Composite models are common when you build reports based on analytical databases (like a Power BI semantic model or an Analysis Services tabular database). These databases already contain an optimized model, calculations in the form of measures, and all sorts of metadata (like hierarchies or translations), which you don't want to rebuild from scratch. Instead, you can connect to such a data source in DirectQuery mode and enrich it by importing other important data sources—and thereby create a composite model.

Not only can a data model have a mixed storage mode, so can a table. Read on to learn more about Dual mode tables.

Dual Mode

When a single query references data from both a table in Import mode and a table in DirectQuery mode, there are two possible ways to resolve this. One way is when, for the sake of the query, all the necessary rows from the table in DirectQuery mode are temporarily loaded into the current model and the joins are resolved inside the model. If you choose DirectQuery mode for a table because of its size, then transferring a large amount of rows from this table into the data model might not be a good idea. It might take time and put pressure on Power BI's resources—that means your computer or the Power BI service.

Therefore, VertiPaq chooses the other option: it pushes all filters to the data source by adding all necessary filters in the query text. This leads to long (and, therefore, probably less performant) queries and potential security risk (see [“Storage Mode” on page 153](#)).

A special storage mode I haven't yet elaborated is a solution here: a table can be set to Dual mode. A table in Dual mode gets refreshed with all other tables that are in Import mode and take up physical space in the data model. Combining information from this table with other tables imported into the data model takes full advantage of the in-memory engine.

At the same time, all necessary meta information is stored in the data model to allow you to reference it in DirectQuery queries. When columns from this table are combined with columns from DirectQuery tables of the same *island*, the generated SQL statement can make use of ordinary joins, instead of injecting filters into the query code. You can expect the query to run faster because all necessary data to satisfy the query is stored in the data source; access to it can be fully optimized (e.g., via indexes).

You change the storage mode in the Model view in the table's properties pane.

Hybrid Tables

Combining imported data and DirectQuery is not only possible within one data model, but also within one table. The idea is to import all the “old” data, which won’t change anymore, but keep all the recent data in DirectQuery mode so it will be shown on reports as it is imported into the data source. Basically, you change the mode per partition of the table.

At the time of writing, you can activate this setting in Power BI Desktop only in the Model view via incremental refresh by enabling “Get the latest data in real-time with DirectQuery (Premium only),” as shown in [Figure 8-12](#). It will automatically keep the latest partition in DirectQuery mode, and the “older” partitions in Import mode.

The concept could also be turned around; you could only load the data for the current reporting period into the data model, but keep the old data, which is just rarely queried, in DirectQuery. Reports covering the standard periods will be fast, but the data model won’t be inflated in terms of size by loading older periods.

In the rare case, when someone needs to report on the old data, the report will work as usual but take more time to execute. This can be a good compromise because people tend to appreciate the fast response for standard use cases and accept that the data for special periods of time aren’t cached. To implement such a flexible concept, you need to [run XMLA scripts against the Power BI service](#).



At the time of writing, hybrid tables are only available in a premium workspaces. As soon as you

Key Takeaways

In this chapter, you learned that there are many options for storing either the data or metadata in Power BI. Combining these options in a smart way can increase the overall performance of the data model by optimizing the trade-off between refresh time and query time. Here are some key takeaways from the chapter:

- Power BI offers three basic storage modes: Import, DirectQuery, and live connection.
 - Imported data offers the best report response time, but the data needs to be refreshed periodically.
 - Neither DirectQuery nor live connection require refreshing the data model, but the queries might run for a longer period of time.

- Live connection is available only for Power BI semantic models and Analysis Services databases. They come with a lot restrictions in terms of data modeling options.
- DirectQuery is available only for relational databases, Power BI semantic models, and Analysis Services databases. Tables in DirectQuery mode can be combined with tables in Import mode in one data model, forming a composite model.
- Partitioning allows you to split a table into smaller parts, which can be refreshed independently. By refreshing partitions only where a change in data has happened, you speed up the overall refresh time of your data model.
- Tables with pre-aggregated contents exchange storage for query runtime, as some queries can be satisfied with the (smaller) table containing the pre-aggregated values. Only if data from a finer granular level is needed is the bigger table queried.
- *Composite model* describes a data model in which storage modes are mixed. Some tables are in Import mode, while others are in DirectQuery mode. (Live connection doesn't allow for a composite model.) Choosing such a model can require a trade-off between storage size and query runtime.
- Dual tables store both the imported data and the metadata to allow for DirectQuery. Such tables allow Power BI to facilitate the full power of the in-memory engine when combined with other imported tables, or inject all logic and join operators into the SQL query when combined with other DirectQuery tables.
- Hybrid tables apply the idea of a composite model to a single table: partitions of a table can be in different storage modes. When real-time access is the goal, you import old data but keep the most recent partitions in DirectQuery mode. If you need to optimize model size, then you load only the most recent data, keeping the old partitions in DirectQuery mode.

Now that you're equipped with all the theoretical concepts and their uses in Power BI, it's time to get you to actively transform your data to fit into the desired data model. **Chapter 9** introduces you to DAX as a data transformation tool. Later chapters explain how to achieve the same results with the help of Power Query/M and SQL.

Data Modeling for Power BI with the Help of DAX

Chapters	Parts				
	Part 1 Data modeling in general	Part 2 Power BI	Part 3 DAX	Part 4 Power Query	Part 5 SQL
Understanding a data model	Chapter 1	Chapter 5	Chapter 9	Chapter 13	Chapter 17
Building a data model	Chapter 2	Chapter 6	Chapter 10	Chapter 14	Chapter 18
Real-world examples	Chapter 3	Chapter 7	Chapter 11	Chapter 15	Chapter 19
Performance tuning	Chapter 4	Chapter 8	Chapter 12	Chapter 16	Chapter 20

Welcome to the DAX part of this book! DAX stands for *data analysis expressions* and is the language used to create calculated tables, calculated columns, and measures; define row-level security; and query the data inside of Power BI. The latter is done for you by Power BI's visuals—so most users will have no need to write a DAX query.

If you have read this book from the start, you should now have a good understanding of the importance of the shape of a data model and the need in many situations (if not all!) to reshape what you get from your data source.

This part is *not* an introduction to the DAX language, but you will learn that data modeling and DAX align very closely. I'll introduce you to the moving parts of the language in [Chapter 9](#):

- Calculated tables
- Actively changing relationships for the sake of a calculation
- Creating a single primary key
- Combining tables in DAX

In [Chapter 10](#), I'll show you how you can use DAX to reshape your data model:

- Normalizing
- Denormalizing
- Adding calculated columns and measures
- Transforming flags and indicators into meaningful text
- Building your own date and time tables
- Implementing role-playing dimensions
- Making the best out of slowly changing dimensions
- Flattening parent-child hierarchies

DAX plays an important role in the real-world use cases, discussed in the first two parts of this book, as well. [Chapter 11](#) will introduce you to DAX solutions for the following use cases:

- Binning
- Multi-fact data models
- Multi-language data models
- Key-value pair tables

Finally, [Chapter 12](#) establishes what you can do with DAX to find a good trade-off between loading data into Power BI or directly querying the data source instead.

Understanding a Data Model from the DAX Point of View

When it comes to Power BI, not all data modeling challenges can be solved in Power Query or the data source. Some data modeling solutions can only be implemented inside a DAX measure. I will point out such solutions in detail where appropriate. It's also important to understand that the data model and DAX work hand-in-hand. This means that in some situations, you can exchange complexity in the Model view for complexity in a DAX measure, or the other way around. You have already seen such situations in [Chapter 7](#). I will point out such situations in this and the following three chapters as well.

But before I dive into the complex examples, I want you to understand how DAX “sees” the data model.

Data Model

The relationship between DAX and the data model is twofold. You can use DAX to add calculated columns to a table and even create whole tables. In addition, DAX uses the information provided by the data model (read: filter relationships) to navigate through the model and access the data needed for the calculation.

To be more precise: whenever you write a formula (or a query) in DAX, it's passed forward to the formula engine, which takes the appropriate steps (requesting values from the storage engine and doing its own calculations in case the storage engine cannot execute the calculations due to complexity). Both the formula and storage engines use all the information available from the data model to do their jobs.

Look at a simple formula: `SUM(Sales[SalesAmount])`. It'll aggregate the `SalesAmount` column of the table `Sales` by adding up all available values. In an unfiltered context (like during evaluation inside a calculated column or in a report where no filter is provided), this will result in the grand total over the content of the whole table. The same calculation will show a different (smaller) number when you provide a filter context, e.g., by adding a slicer on the report and selecting a certain year or product. This will be no surprise because this is the expected, and quite intuitive, behavior.

In “[Data Model](#)” on page 75, you learned that you shouldn't put a filter directly on a fact table (the `Sales` table, for example) but on the dimension tables, like `Date` or `Product`. If you have correctly defined the filter relationships between the dimension tables and the fact table, these filters are automatically propagated from the dimension table to the fact table for you. The information defined in the model helps to write rather simple DAX formulas—we don't need to repeatedly specify that the `Sales` table needs to be filtered by the other tables in every single formula but must define this only once in the data model in the Model view.

A data model consists of tables and relations between them. Let's first look at how you can use and create tables in DAX.

Basic Components

The following basic components are important to understand as you learn more about DAX: tables, relationships, and primary keys.

Tables

In DAX, you can refer to a table by just mentioning its name. If the table's name is a reserved keyword, matches a DAX function's name, or contains special characters (like a space or an umlaut), then you must specify the table's name within single quotes. Common table names that require single quotes are `'Date'` and `'Product'`. I tend to specify single quotes every time I remember to do so, to make my code conform better. That means I refer to `'Sales'` (including the single quotes, even when optional) instead of just `Sales` (which would syntactically be no problem). Look up [DAX syntax](#) if in doubt about the need for single quotes or other syntax-related questions.

To create a table in DAX, choose Modeling → “New table.” Then, you have several options to specify a DAX code, which must be a table expression. Examples [9-1](#), [9-2](#), [9-3](#), [9-4](#), and [9-5](#) give you an idea.¹

¹ The examples in this section use the [DAX table.pbix](#) file from the book's GitHub repository.



By just looking at a piece of DAX code, it can be difficult to recognize whether it's the definition of a calculated column, a measure, or a calculated table. I use the following conventions:

```
[Measure Name] :=  
    <definition of a measure>  
  
'Table Name'[Column Name] =  
    <definition of a calculated column>  
  
[Table Name] = /* calculated table */  
    <definition of a calculated table>
```

Example 9-1. Use braces (“{}”) to specify a table expression, returning a row.

```
My 1st table = /* calculated table */  
{  
    1,  
    "one"  
}
```

Value
1
one

Example 9-2. Use parentheses (“()”) inside of braces (“{}”) to specify a table expression, containing several rows.

```
My 2nd table = /* calculated table */  
{  
    (20, "twenty"),  
    (21, "twentyone")  
}
```

Value1	Value2
20	twenty
21	twentyone

Example 9-3. The function ROW() allows you to specify pairs of column names and expressions. This creates one row.

```
My 3rd table = /* calculated table */  
ROW(  
    "My first column", 3,  
    "My second column", "three"  
)
```

My first column	My second column
3	three

Example 9-4. For function DATATABLE(), you can pass parameters to specify both the data type and the names of the columns in one go.

```
My 4th table = /* calculated table */
DATATABLE(
    "MyNumber", INTEGER,
    "MyText", STRING,
    {
        {40, "forty"},
        {41, "fortyone"}
    }
)
```

MyNumber	MyText
40	forty
41	fortyone

Example 9-5. The simplest table expression is created by typing the name of another existing table. This way, you duplicate the full content of the table under a new name.

```
Referenced table = 'My 1st table'
```

Value
1
one

I use the table expression `Measure = {BLANK()}` as a shortcut to create a dedicated (calculated) table that contains all my measures (if the report creators request to have measures collected in one place in the data model, as opposed to them being spread out across the data model's different tables). This expression creates a table with one column (with the name `Value`) and one row (containing *blank*). You can achieve the same by providing only the name of the calculated table and an equal sign but omitting the expression `Measure =`.

Don't forget to hide the only column of this table after you create your first measure. When a table contains only measures and all columns are hidden, the table becomes listed on the top of Power BI's field list in the Power BI Desktop's Data pane.

References to columns should always mention the table name where the column resides. The column's name must be enclosed in brackets (`[]`), whether or not it

contains special characters. For example, `SUM(Sales[SalesAmount])` passes the reference to column `SalesAmount` of table `Sales` as the parameter for function `SUM`.

Tables don't exist by themselves; they exist in relation to other tables (see “[Tables](#)” on [page 6](#)).

Relationships

In most cases, you will not specify any relationships in DAX but rather rely on what you already configured in the Model view of Power BI. This is because you should create all relationships in the Model view so that Power BI can index these relationships. In [Chapters 10](#) and [11](#), I explain situations in which creating a relationship inside a DAX measure instead is necessary.

One of these use cases occurs when you have inactive relationships (see “[Relationships](#)” on [page 88](#)). DAX allows you to activate an inactive relationship just for the sake of the current measure. To achieve this, you need to wrap the expression inside the `CALCULATE` function and add `USERELATIONSHIP` (pronounced *use relationship*) as the second parameter.² `USERELATIONSHIP` itself is a function as well. Via the two parameters, you can specify the names of the columns used in the inactive relationship:

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

DAX is smart enough to then deactivate the active relationship implicitly, so only one relationship is active during the evaluation of the calculation.

With function `TREATAS` (pronounced *treat as*), you can create a relationship inside a DAX formula that isn't available in the Model view. Such a relationship is called a *virtual relationship*. This can be helpful in certain use cases but can incur a performance penalty—the engine can't rely on pre-calculated information about how to join the tables (as it can do with explicitly defined relationships in the Model view). Therefore, use `TREATAS` with care, and instead, rely as much as possible on (physical) relationships in the data model. (I use `TREATAS` in real-world scenarios in “[Budget](#)” on [page 220](#).)

The only way to avoid the need to use `USERELATIONSHIP` and `TREATAS` is to build your data model in a different way, so DAX can rely on active physical relationships. If you decide against such active physical relationships, one or the other function can be

² The examples in this section use the *Relationship.pbix* file from the book's GitHub repository.

very helpful. Both functions are useful for situations in which you need to write your DAX measures in a certain way; this can't be substituted for Power Query or SQL (unless you use Power Query or SQL to create a data model that can fully rely on active physical relationships).

Relationships in a data model are based on primary and foreign keys. DAX is no exception.

Primary Keys

In DAX, there's no explicit concept of primary keys. As described in the previous section, you should define all relationships in the Model view. Then, you can explore the the relationships between the tables (which preferably have a one-to-many cardinality and, therefore, implicitly describe a relationship between a primary key on the "one" side and a foreign key on the "many" side). These relationships can be created on only a single column; composite keys are not supported in Power BI.

Composite primary (and foreign) keys must be converted into one key with the help of the CONCATENATE function or the & operator. CONCATENATE only allows for a pair of parameters, so you need to wrap multiple calls to this function to achieve what you can do with a shorter piece of code with the & operator:

```
CustomerKey =  
    CONCATENATE(  
        Customer[Firstname],  
        CONCATENATE(  
            "|",  
            Customer[LastName]  
        )  
    )  
CustomerKey = Customer[Firstname] & "|" & Customer[LastName]
```

One way to combine tables is to append one table to another. This is the basic idea of set operators.

Combining Queries

Combining queries can be accomplished in two ways: either you use set operators or you use join operators. You can use the *Functions Relational.pbix* file to follow along with the examples in this section.

Set Operators

DAX offers functions for the usual set operations (see "Set Operators" on page 11). For the examples in this section, I use the Product table, which contains a List Price column, and the Sales table, which contains a Price column, as shown in Tables 9-1 and 9-2.

Table 9-1. Product

Product ID	List Price
100	10
110	30
120	110
130	200

Table 9-2. Sales

Date	Product ID	Price
2023-08-01	100	10
2023-08-01	110	20
2023-08-02	110	30
2023-08-03	120	100

UNION

The UNION function allows for multiple parameters. Duplicated rows are not removed. To keep only unique rows, you would need to wrap UNION into DISTINCT.

In [Table 9-3](#), you can see the result of an example where I apply the function to the available VALUES of the product's List Price and the Sales table's Price column inside the definition of a calculated table. The List Price of 10 appears twice in the result:

```
UNION = /* calculated table */  
UNION(  
    VALUES('Product'[List Price]),  
    VALUES(Sales[Price])  
)
```

Table 9-3. UNION

List Price
10
30
110
200
10
20
30
100

INTERSECT

INTERSECT returns only rows, which appear in both tables. The following example results in the rows shown in [Table 9-4](#):

```
INTERSECT = /* calculated table */  
INTERSECT(  
    VALUES('Product'[List Price]),  
    VALUES(Sales[Price])  
)
```

Table 9-4. INTERSECT

List Price
10
30

EXCEPT

EXCEPT returns only rows, which appear in the first table but not in the second. The result of the following code is shown in [Table 9-5](#):

```
EXCEPT = /* calculated table */  
EXCEPT(  
    VALUES('Product'[List Price]),  
    VALUES(Sales[Price])  
)
```

Table 9-5. EXCEPT

List Price
110
200

A different way of combining tables is to synchronize their rows and create a query result, which consists of the combined list of columns for both of the tables.

Joins

In most scenarios, you'll rely on the existing tables (and their relationships) in the data model. In certain use cases, it could make sense to create a calculated table (to persist the result in the data model to speed up queries) or use a table expression inside a measure (to solve advanced use cases by explicitly joining tables and iterating over the result).

I use the two tables, Product and Sales (Tables [9-1](#) and [9-2](#)), from the previous section here to look at the DAX functions you can use.

NATURALLEFTOUTERJOIN

NATURALLEFTOUTERJOIN applies a left outer join to the specified two tables. This function comes in handy when you decide against creating an active relationship in the Model view. Usually, as mentioned previously, you'd explicitly create an active relationship between those two tables in the data model.

If a relationship does exist, you can still join the two tables with NATURALLEFTOUTERJOIN but must first break the data lineage of the tables by manipulating the key columns (e.g., adding 0 or concatenating an empty string to the key columns). Keep in mind that you can't specify the join predicate (the names of the columns to use for the join operation) in a natural join—therefore, the key columns have to have the exact same name in both tables. *Left outer* means that all the rows from the first (left) table are kept, and information from matching rows of the second table are added. This join is implemented as an equi-join.

Here, I put the Sales table as the first (left) table. I need to wrap the reference to the Product table inside ALLEXCEPT to get rid of its Product ID column because the result would otherwise contain two Product ID columns, which is not allowed:

```
NATURALLEFTOUTERJOIN Sales = /* calculated table */
NATURALLEFTOUTERJOIN(
    'Sales',
    ALLEXCEPT('Product', 'Product'[Product ID])
)
```

And Table 9-6 shows the result.

Table 9-6. NATURALLEFTOUTERJOIN Sales

Date	Product ID	Price	List Price
2023-08-01	100	10	10
2023-08-01	110	20	30
2023-08-02	110	30	30
2023-08-03	120	100	200

If I exchange the order of the two tables inside NATURALLEFTOUTERJOIN, the result will be slightly different. First, the orders of the columns in the result are exchanged. And second, because the Product table contains a Product ID for which there are no entries in the Sales table, the number of rows changes. When the Product table is on the right side of a left outer join, this row is omitted from the result. When the Product table is on the left side of a left outer join, then this row is kept and shown in the final result:

```
NATURALLEFTOUTERJOIN Product = /* calculated table */
NATURALLEFTOUTERJOIN(
    ALLEXCEPT('Product', 'Product'[Product ID]),
```

```

    'Sales'
)

```

And **Table 9-7** shows the result.

Table 9-7. NATURALLEFTOUTERJOIN Product

List Price	Date	Product ID	Price
10	2023-08-01	100	10
30	2023-08-01	110	20
30	2023-08-02	110	30
200	2023-08-03	120	100

NATURALINNERJOIN

NATURALINNERJOIN applies a natural inner join on the specified two tables. All the rules about relationships and data lineage apply here as well. As this is an inner join, only rows with matching keys in both tables are kept. This join is implemented as an equi-join.

In our example, the result of the inner join matches the result of the left outer join when the sales table was on the left side, as you can see:

```

NATURALINNERJOIN = /* calculated table */
NATURALINNERJOIN(
    'Sales',
    ALLEXCEPT('Product', 'Product'[Product ID])
)

```

And **Table 9-8** shows the result.

Table 9-8. NATURALINNERJOIN

Date	Product ID	Price	List Price
2023-08-01	100	10	10
2023-08-01	110	20	30
2023-08-02	110	30	30
2023-08-03	120	100	200

CROSSJOIN

CROSSJOIN creates the Cartesian product of the two specified tables (**Table 9-9**). This also works in a table (expression) where no physical relationship was created in the data model:

```

CROSSJOIN = /* calculated table */
CROSSJOIN(
    DISTINCT('Sales'[Date]),

```

```
    DISTINCT('Product'[Product ID])
)
```

Table 9-9. *CROSSJOIN*

Date	Product ID
2023-08-01	100
2023-08-02	100
2023-08-03	100
2023-08-01	110
2023-08-02	110
2023-08-03	110
2023-08-01	120
2023-08-02	120
2023-08-03	120
2023-08-01	130
2023-08-02	130
2023-08-03	130

I return to these functions in “[Denormalizing](#)” on page 190 and show how to use them to denormalize a data model.

The whole idea of combining tables in this book is to bring them into a different shape to create the perfect data model (a star schema). But there’s much more you will need to do (namely, ETL).

Extract, Transform, Load

All the necessary ETL steps are ideally done before the data is loaded into Power BI (see the “Extract, Transform, Load” sections of Chapters 13 and 17). DAX is less ideal for implementing the ETL process because it can only build on top of tables that are already loaded into the data model. If you use DAX to model the data, then your model will contain both the un-modeled data and the modeled data, thus unnecessarily increasing the size of your data model.

You can find the size of a model by checking the size of the *.pbix* file, but it also has to do with how much memory the data model will occupy when you open *.pbix* in Power BI Desktop or when an Analysis Services tabular database is refreshed.

But cleaning, transforming, and modeling your data in DAX (see [Chapter 10](#)) is better than not modeling the data at all.

Key Takeaways

In this chapter, you learned about important moving parts for creating a data model in the DAX language. Specifically, you now know the following things:

- DAX can handle tables as expressions and parameters. You can create a calculated table in DAX if needed.
- With DAX, you can create not only calculated tables but calculated columns as well.
- Relationships are usually not explicitly maintained in a DAX expression, but the definition of the filter relationships (created in the Model view) is implicitly in effect in every DAX expression. Functions like `USERELATIONSHIP` or `TREATAS` allow you to explicitly change relationships inside a DAX expression.
- Set operators `UNION`, `INTERSECT`, and `EXCEPT` are available as DAX functions.
- DAX offers functions to implement a natural join (`NATURALINNERJOIN` and `NATURALLEFTOUTERJOIN`) and a cross join (`CROSSJOIN`).
- You should push the ETL process to earlier stages in your data platform architecture. DAX should be used only as a last resort to clean and transform data (it can't substitute the uncleaned and untransformed data but only add the cleaned and transformed version to the data model). The only thing you can't do in the previous stages without using DAX are non-additive calculations—that's where we need DAX and where it shows its true power.

Now that you have a basic understanding of how the data model and DAX interact with each other, it is time to learn how you can actively shape the data model with the help of the DAX language in the next chapter of this book.

Building a Data Model with DAX

With DAX you are writing *data analysis expressions*, which allow you to create calculated tables, calculated columns, and, most importantly, measures (and row-level security and queries, which aren't in the scope of this book). Everything you can achieve with calculated tables and columns, you can also achieve with solutions in Power Query/M and with SQL. If you just started with Power BI, then you need to learn DAX anyway; some problems can only be solved with measures written in DAX—and you might implement the transformations to build your data model in DAX as well.

Normalizing

As Chapters 1 and 2 detail, normalizing is important for fact tables and means that you strip the table of replicated information. You only keep foreign keys to one or more tables, which contain DISTINCT lists of the otherwise redundant information. These other tables are the dimension tables.

With that said, normalizing is as easy as removing all columns with repeated information that don't comprise the (primary) key of the information and putting them into a table of their own. To find out which columns contain repeated information, I create a table visual in Power BI with a single column or a combination of columns that might have one-to-one relationships with each other.¹ Power BI will automatically show only the distinct values.

¹ The examples in this section use the *Normalize Facts DAX.pbix* file from the book's GitHub repository.

In **Figure 10-1**, the combinations of columns listed here are candidates for dimensions:

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

Discount Band	Segment	Country
High	Channel Partners	United States of America
Low	Enterprise	Mexico
Medium	Government	Germany
None	Midmarket	France
	Small Business	Canada

Product	Manufacturing Price	Date	Month Name	Month Number	Year
Amarilla	260	2013-09-01	September	9	2013
Carretera	3	2013-10-01	October	10	2013
Montana	5	2013-11-01	November	11	2013
Paseo	10	2013-12-01	December	12	2013
Velo	120	2014-01-01	January	1	2014
VTT	250	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 10-1. Dimension candidates

Always also use your domain knowledge and discuss with the domain experts to decide if the candidates you select are indeed good choices. Especially if you only work with demo or test data (and not production data), the true relationships among the columns might not be clear from just looking at the available data.

When you have agreement that your candidates 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're asking yourself if it's really worth it to create such 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](#)? You must avoid direct filters on the fact table under all circumstances.



Adding attributes directly into the fact table is sometimes recommended. Order numbers are a typical example where it isn't a good idea to create a dedicated dimension table. The cardinality of the dimension table would be close to the fact table, and the size of the fact table wouldn't be reduced because, instead of the order number, you'd need to add the foreign key to the fact table.

Be aware that adding such information to the data model comes at a price: the size of the Power BI semantic model will increase dramatically as a column of a high cardinality compresses badly. Propagating a slicer with the column values will take a while, and applying the filter in a visual won't be fast. On top of all this, you could suffer from the problem pointed out in [“A Single Table to Store It All”](#) on page 103.

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., weekends, bank holidays, etc.), which you can clearly see in [Figure 10-1](#). I show how to create a fully functional date table later in [“Time and Date”](#) on page 203.

We can't, however, physically remove any column from the fact table, but only hide those columns (so that the report creators are not unintentionally using them)—otherwise, creating the calculated tables (referencing those columns) would fail.



There's a big disadvantage to 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 its original (un-modeled, non-star) shape.

You can't truly transform the data into the intended model, but that shouldn't keep you from applying these best practices. It's better to shape your data with DAX than not shape it at all. Just remember that when the size of the data model (in memory—but you can also just take a look at the size of the *.pbix* 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 refactor your DAX into Power Query (or SQL).

Denormalizing

I'm sure, by now, you're aware that we need to denormalize dimension tables. To denormalize, we need to add columns with information from a related table into the main dimension table. The DAX function that 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 this next example, a product's information is split into three tables: `DimProduct`, `DimProductSubcategory`, and `DimProductCategory`.² Simply create two new calculated columns in table `DimProduct`:

```
DimProduct[Subcategory] =  
    RELATED(DimProductSubcategory[EnglishProductSubcategoryName])  
DimProduct[Category] =  
    RELATED(DimProductCategory[EnglishProductCategoryName])
```

Because there's 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 from `DimProduct` over `DimProductSubcategory` to `DimProductCategory` as well. 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 report creators using these columns unintentionally, but we cannot actually delete the two tables from the model (because then, the newly created calculated columns would throw an error).

Calculations

Calculations are the home game for DAX. DAX is built for creating formulas for even very complex challenges. And by “calculations,” I mostly mean explicit measures; that's the core competence of a data analytic expression. Creating calculated tables and calculated columns is possible as well, but I see the value of these functions only historically, as a workaround from the early days of Excel's Power Pivot, when Power Query wasn't available yet. In many scenarios, you're better off with explicit measures as opposed to using calculated columns or adding columns in Power Query or SQL.

Before I dive into kinds of calculations, let's see how many resources you can save by replacing redundant columns (those whose value can be calculated by using other

² This example uses the *Denormalize Dimension DAX.pbix* file.

existing columns) with a DAX measure. In the *Financials Filter Dimension Surrogate Key Measures.pbix* file, you'll find the following three redundant columns:

- Gross Sales can be calculated from the Sale Price and Units Sold, as follows:

```
Gross Sales :=  
SUMX(  
    financials,  
    financials[Sale Price] * financials[Units Sold]  
)
```

- Sales can be calculated from the Gross Sales and Discounts, as follows:

```
Sales := [Gross Sales] - SUM(financials[Discounts])
```

- Profit can be derived from Sales and COGS ("cost of goods"), as shown here:

```
Profit := [Sales] - SUM(financials[COGS])
```



In DAX, every use of the SUM function can be rewritten as SUMX—that's what the storage engine does. There's no difference in performance, only in the syntax. If you want to add up the value of a column, use SUM (e.g., SUM(financials[Discounts])). If you need to add the result of an expression, you need to use SUMX:

```
SUMX(financials,  
    financials[Sale Price] * financials[Units Sold])
```

SUM doesn't allow you to provide an expression, only a column reference.

If you replace the existing columns in the Financials table with the explicit DAX measures, you will recognize a remarkable difference in the sizes of the data models:

- A model with the Financials table as is uses 5.5 MB of RAM (see *Financials OBT.pbix*).
- A model where I replaced the three columns Gross Sales, Sales, and Profit with the aforementioned DAX measures only occupies 253 KB of RAM (see *Financials OBT Measures.pbix*).

Getting rid of the columns by replacing them with mathematically identical DAX measures reduced the size of the model to a 20th of its size—without losing any information or features in the data model. Instead of persisting the results of these calculations (and occupying space), they're replaced with their formula, which will be calculated as needed when queried.

But what about query performance, you ask? Turns out that the query plan for the two models isn't as different as you might expect. To measure it, I create a visual with

the three columns/measures organized by date in each of the two Power BI files. The two reports look very similar: they show the same numbers, although I keep the default names for the headers (“Sum of ...”) to make it easy to distinguish the two reports (Figures 10-2 and 10-3).

For each file, I then switch on View → Performance Analyzer, then hit “Start recording” and “Refresh visuals.” If you then expand Table, you can click “Copy query.” Next, I start DAX Studio (via “External tools”), paste the query, and switch on “Query plan,” “Server timings,” and “Clear on Run” via the ribbon. Then, you can run the query.

Because both data models are very small, the queries will finish in no time. But I want to point out that the storage engine query is almost identical. Here, you see the xSQL representation (choose Server Timings below the query and click the only line to display the query) of what the storage engine needs to do to deliver the rows for the report for the data model where all columns are persisted. It simply sums up the stored values for the columns Profit, Gross Sales, and Sales:

```
SET DC_KIND="AUTO";
SELECT
    'LocalDateTable'[Year],
    'LocalDateTable'[MonthNo],
    'LocalDateTable'[Month],
    'LocalDateTable'[QuarterNo],
    'LocalDateTable'[Quarter],
    'LocalDateTable'[Day],
    SUM ( 'financials'[Profit] ),
    SUM ( 'financials'[Gross Sales] ),
    SUM ( 'financials'[ Sales] )
FROM 'financials'
LEFT OUTER JOIN 'LocalDateTable'
    ON 'financials'[Date]='LocalDateTable'[Date];
```

Estimated size: rows = 16 bytes = 576

Year	Quarter	Month	Day	Sum of Gross Sales	Sum of Sales	Sum of Profit				
2013	Qtr 3	September	1	4.729.736,00	4.484.000,03	763.603,03				
2013	Qtr 4	October	1	9.828.688,00	9.295.611,10	1.657.795,10				
2013	Qtr 4	November	1	8.167.338,00	7.267.203,30	765.502,30				
2013	Qtr 4	December	1	5.835.025,00	5.368.441,08	691.564,08				
2014	Qtr 1	January	1	7.307.403,50	6.607.761,68	814.028,68				
2014	Qtr 1	February	1	7.699.201,00	7.297.531,39	1.148.547,39				
2014	Qtr 1	March	1	6.124.026,00	5.586.859,87	669.866,87				
2014	Qtr 2	April	1	7.429.392,50	6.964.775,07	929.984,57				
2014	Qtr 2	May	1	6.767.911,00	6.210.211,06	828.640,06				
2014	Qtr 2	June	1	10.268.972,00	9.518.893,82	1.473.753,82				
2014	Qtr 3	July	1	8.833.027,50	8.102.920,18	923.865,68				
2014	Qtr 3	August	1	6.325.959,00	5.864.622,42	791.066,42				
2014	Qtr 3	September	1	6.845.317,00	6.398.697,24	1.023.132,24				
2014	Qtr 4	October	1	13.313.424,00	12.375.819,92	1.781.985,92				
2014	Qtr 4	November	1	5.947.910,00	5.384.214,20	604.600,20				
2014	Qtr 4	December	1	12.508.268,00	11.998.787,90	2.025.765,90				
Total				127.931.598,50	118.726.350,26	16.893.702,26				

Figure 10-2. Report based on persisted columns

Year	Quarter	Month	Day	Gross Sales	Sales	Profit				
2013	Qtr 3	September	1	4.729.736,00	4.484.000,03	763.603,03				
2013	Qtr 4	October	1	9.828.688,00	9.295.611,10	1.657.795,10				
2013	Qtr 4	November	1	8.167.338,00	7.267.203,30	765.502,30				
2013	Qtr 4	December	1	5.835.025,00	5.368.441,08	691.564,08				
2014	Qtr 1	January	1	7.307.403,50	6.607.761,68	814.028,68				
2014	Qtr 1	February	1	7.699.201,00	7.297.531,39	1.148.547,39				
2014	Qtr 1	March	1	6.124.026,00	5.586.859,87	669.866,87				
2014	Qtr 2	April	1	7.429.392,50	6.964.775,07	929.984,57				
2014	Qtr 2	May	1	6.767.911,00	6.210.211,06	828.640,06				
2014	Qtr 2	June	1	10.268.972,00	9.518.893,82	1.473.753,82				
2014	Qtr 3	July	1	8.833.027,50	8.102.920,18	923.865,68				
2014	Qtr 3	August	1	6.325.959,00	5.864.622,42	791.066,42				
2014	Qtr 3	September	1	6.845.317,00	6.398.697,24	1.023.132,24				
2014	Qtr 4	October	1	13.313.424,00	12.375.819,92	1.781.985,92				
2014	Qtr 4	November	1	5.947.910,00	5.384.214,20	604.600,20				
2014	Qtr 4	December	1	12.508.268,00	11.998.787,90	2.025.765,90				
Total				127.931.598,50	118.726.350,26	16.893.702,26				

Figure 10-3. Report based on measures to calculate the values on the fly

When I remove those three columns and add explicit measures to the data model, the xmSQL query is slightly changed: the result for the three KPIs (Gross Sales, Sales, and Profit) cannot be obtained from the data. Instead, the values for columns Discount and COGS are summed up, an expression to multiply the Sales Price

column by the Units Sold column is added, and its result is summed up (to satisfy the calculations for the three measures; it's performed by the formula engine):

```
SET DC_KIND="AUTO";
WITH
    $Expr0 := ( CAST ( PFCAST ( 'financials'[Sale Price] AS INT ) AS REAL )
        * PFCAST ( 'financials'[Units Sold] AS REAL ) )
SELECT
    'LocalDateTable'[Year],
    'LocalDateTable'[MonthNo],
    'LocalDateTable'[Month],
    'LocalDateTable'[QuarterNo],
    'LocalDateTable'[Quarter],
    'LocalDateTable'[Day],
    SUM ( 'financials'[Discounts] ),
    SUM ( 'financials'[COGS] ),
    SUM ( @$Expr0 )
FROM 'financials'
LEFT OUTER JOIN 'LocalDateTable'
    ON 'financials'[Date]='LocalDateTable'[Date];
```

Estimated size: rows = 16 bytes = 576

In the end, the query performance is identical (and the random differences with every execution are bigger than the differences between the two queries measured with only one run). Your mileage may vary with more complex calculations.

The difference in storage space is not so much about the three decimal values but the cardinality of these three columns:

- Gross Sales contains 550 distinct values.
- Sales contains 545 distinct values.
- Profit contains 557 distinct values.

Power BI's in-memory compression algorithm is highly dependent on the cardinality of a column. As the whole fact table has 700 rows, we can derive that almost every row has a different Gross Sales, Sales, and Profit. The high cardinality of these columns leads to the high amount of space these columns occupy in RAM, even after compression.

Conversely, refresh time might improve in the model wherein you don't add the three columns to the data model (it doesn't have to be calculated in the data source and less data has to be moved when loaded into the data model).

Overall, you should consider explicitly creating measures for all types of calculations (including the aforementioned columns Discount and COGS, which I wrapped in DAX's SUM function).

Simple Aggregations for Additive Calculations

Simple aggregations for additive calculations could also be calculated through Default Summarization. **Part II** explains 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 makes 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 calculates 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 Price]  
    )
```

Semi-Additive Calculations

Semi-additive calculations require you to specify for which date the value should be calculated. Usually, it's 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-measures-in-dax/  
    */  
    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-measures-in-dax/
*/
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

```



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

Re-create the Calculation as a DAX Measure

Results of non-additive calculations simply can't be aggregated in a meaningful way. 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 row in the sales table. But a report barely shows the individual sales rows, instead showing 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('Sales'[Sales]))

```

This measure also works on the level of individual sales, where only a single sale event is available (because 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 of 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. Some customers might have bought something in more than one month. You should ensure that those customers aren't counted twice when calculating the DISTINCT count for the quarter. But creating such a measure is not a big deal:

```
[DISTINCT Count of Products] := DISTINCTCOUNT('Sales'[Product ID])
```

You can see in [Figure 10-4](#) 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 (C) of the month. But in total, there've been only three different products (A, B, and C) sold during those three days (product B was sold on both the first and 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 complain on social media that the table visual in Power BI is buggy because it doesn't always add up the individual numbers in the total. It clearly depends on the context of a calculation, whether the individual numbers need to be aggregated or if the calculation has to be done on the aggregated level (see [Part I](#)).

Sales						
Date	Product	Price	Quantity	Date	Count of Products	Distinct Count of Products
01.09.2021	A	10	3	01.09.2021	2	2
01.09.2021	B	20	1	02.09.2021	1	1
02.09.2021	B	30	4	03.09.2021	1	1
03.09.2021	C	100	5	Total	4	3

Figure 10-4. Visual showing measures Count of Products and Distinct Count of Products



Non-additive calculations, like a distinct count, 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

Time-intelligence calculations are another use case that 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.

Explicit measures

You can use DAX's built-in functions either to directly calculate the value (e.g., `TOTALYTD`) or as filter parameters for `CALCULATE` (e.g., `DATESYTD`). Those functions hide some complexity from you, but you can always come up with a formula that achieves 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 semantics, 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 10-5 shows the identical result of the three different approaches in DAX to calculate the year-to-date (YTD) total.

Sales				Date	Sales Amount	TOTALYTD	TOTALYTD 2	TOTALYTD 3
Date	Product	Price	Quantity					
01.09.2021	A	10	3	01.09.2021	50	50	50	50
01.09.2021	B	20	1	02.09.2021	120	170	170	170
02.09.2021	B	30	4	03.09.2021	500	670	670	670
03.09.2021	C	100	5	04.09.2021		670	670	670
				Total	670			

Figure 10-5. Results of the three measures to calculate the year-to-date value for *Sales Amount*



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

Calculation groups

Requirements for time intelligence, especially, 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 tedious to create (and maintain) all the variations for each measure. Here, *calculation groups* come in 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., YTD for a measure as one 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 measure's name (e.g., [Sales Amount]) with the function SELECTEDMEASURE:

```
[Actual] := SELECTEDMEASURE()
```

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

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 the time of writing, Power BI Desktop's UI doesn't expose its definitions. Therefore, you need to use a third-party tool to create and maintain calculation groups in your .pbix file. If you're working with Analysis Services tabular, you have full access to the definition of calculation groups in, for example, Visual Studio.

Figure 10-6 shows Tabular Editor 3, but you could use the free version of Tabular Editor (version 2) to maintain calculation groups. In the first step, you would need to create a new calculation group by right-clicking Tables inside TOM Explorer. I renamed both the table and column, from "Name" to "Time Intelligence." Then, add

Calculation Items per variance. Here, I added one for Actual and one for YTD as described.

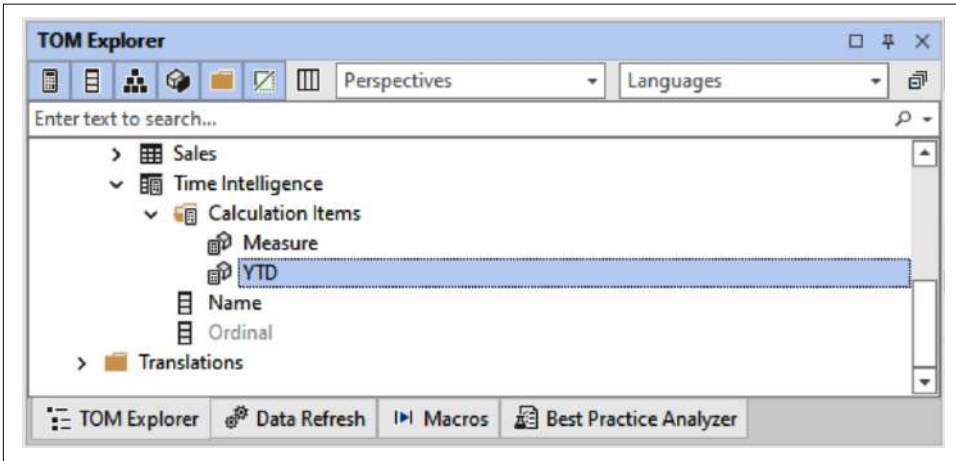


Figure 10-6. Defining a calculation group in Tabular Editor 3

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. Use the *Flag.pbix* file to follow along with this section's examples.

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. If the first parameter does not evaluate to TRUE, the third parameter is returned. 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 a simple list of values, which I prefer over nested IFs due to the better readability. For calculated column `Product Line Description`, I did provide a column name as the first parameter (`DimProduct[ProductLine]`) and different literals for the even (second, fourth, etc.) parameters ("R", "M", "T", and "S").

If the first parameter matches one of these literals, then one of the odd (third, fifth, etc.) parameter values is returned (either "Road", "Mountain", "Touring", or "Standard"). I provide a last parameter ("other") for cases where a new `Product Line` might be added after I write the formula. If I 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 use `TRUE`, and the even parameters each 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 (which 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 that's returned when all the comparisons fail:

```
'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 maintain the content of a table than to rewrite 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, a lookup has benefits as well (as we discuss in “[Multi-Language Model](#)” on page 63). Creating the lookup table in DAX is clearly not my favorite method (because changing the content of the table means a change to the formula of the calculated table), but it can be done with the DATATABLE function. The following code shows how to use function DATATABLE to create a table called Styles (DAX):

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

Then, you create a filter relationship between the table’s Styles (DAX) column Style and the table’s DimProduct column Style. This enables you to use RELATED to look up the values. In case a value for Style that is present in DimProduct isn’t (yet) available in table Styles (DAX), I check for BLANK and return unknown:

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

Treating BLANK values

Sometimes you don’t need to develop a 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 databases call those missing values NULL.

You can either compare an expression against `BLANK()` or explicitly check if an expression is blank by passing the expression into function `ISBLANK`. In calculated column `WeightUnitMeasureCode`, I replace empty and blank values with `N/A`:

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

Time and Date

As [Chapter 6](#) points out, 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 adapt to the needs of your report users.³ The number of rows in a `Date` or `Time` table is usually negligible—so you don’t have to limit yourself in terms of 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 ranges your fact tables contain. Basically, you 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 of the most recent year. This will work as long as you don’t import columns with “exotic” dates (like birth dates or placeholders like January 1, 1900 or December 31, 9999). In case of fiscal 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 table */
```

CALENDAR

The `CALENDAR` function gives you more control; 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 isn’t very flexible (and requires you to remember 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

³ See *Date.pbix* in the book’s GitHub repository to follow along with the examples in this section.

be fast enough and give you peace of mind—you'll know 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(MAX('Fact Reseller Sales'[OrderDate])),
        12, /* December */
        31 /* 31st */
    )
)
```

After creating the calculated table, you can add new columns over Power BI Desktop's UI. However, I recommend that you nest CALENDARAUTO or CALENDAR into ADDCOLUMNS and then specify pairs of names and expressions for the additional columns. With that approach, you'll have everything in one 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(MAX('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()), "Current Year", YEAR([Date]))
)
```

There are several additional columns typically used for a date table:

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 multiply by 10,000, add the number of the month multiplied by 100, and then add the day. In a data warehouse, it's best practice to also have the keys for dates in the form of a whole number. In Power BI and Analysis Services tabular, this isn't as important.

Year as the year portion of the date

The DAX function YEAR has you covered here.

MonthKey, Month Number, Month, YYYY-MM

Variations of the month, like the month number of the year, the month name, the year and month combined in different formats. Most of the variations can be calculated by using function FORMAT and passing in a format string.

Weeknumber (ISO)

You pass the date as the first parameter for function WEEKNUM. The second parameter allows you to specify whether your week starts on Sundays or Mondays, or if the week number should be calculated according to the ISO standard.

Current Year

Users expect that a report shows the most recent data. Pre-selecting the right year and month can be challenging 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, you can add additional columns to this expression, so we have the full definition of this calculated table in one place:

- The TIME function can convert the pairs of hours and minutes into a proper column of datatype Time .
- The FORMAT function can also do wonders with time-related content:

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

```
) "Time Description", FORMAT(TIME([Hour], [Minute], 0), "HH:MM")
```

Role-Playing Dimensions

If you opt to load a role-playing dimension into a data model only once, you need to make sure that you add as many relationships as foreign keys from one table to the existing role-playing dimension. A maximum of one of those relationships can be active; the others can only 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 can create individual measures like Order Quantity, Sales Quantity, etc. Each of these measures uses 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 only one relationship remains active at a given point in time:⁴

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



Calculation groups can help with role-playing dimensions. 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 Date table, you 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 if the size of the role-playing dimension isn't huge, the increase should be negligible. The advantage is that you don't need to create variations of your measures (by applying CALCULATE and USERELATIONSHIP). The report creator chooses one copy of the dimension table over the other—or even combines both.

⁴ The examples in this section use the *Date role-playing.pbix* file from the book's GitHub repository.

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. But I 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 changing the expression referring to just the base table and use `SELECTCOLUMNS`, which allows you to specify which columns (or expressions) you want to return under which column name:

```
[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]
)
```

This approach allows you to again have all the logic (renaming) in one place (namely, the expression for the calculated table). In Parts [IV](#) and [V](#), I show how you can automatically rename all columns without specifying each and every column, as we need to do in DAX.

Slowly Changing Dimensions

Slowly changing dimensions must be implemented in a physically implemented data warehouse. In DAX, you can't update rows to keep track of changes and different versions, only load the results of such updates.

Usually, a slowly changing dimension doesn't require extra effort in the world of DAX; the rows in the fact table already reference the right version of the dimension table. Only if your report user needs to override the default version (the version that was valid at the point in time the fact was collected) would you need to reach out to DAX and implement the logic via `CALCULATE` in your measures.

Figure 10-7 shows a report page with the following content:⁵

- A slicer to choose the product.
- A slicer to choose the year, month, or day when the product has 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 at the top of the screen:
 - Selected Product Version shows the latest available date for the year selected in the Product Version slicer. For the year 2023, this is 2023-12-31.
 - Just beneath that is the Product Version's Standard Cost for the product on that day. On December 31, 2023, the standard cost was €39.2589.
- 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 Version's Standard Cost, as shown at the top of the screen times the Quantity sold on that day

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). At the beginning of 2022, the StandardCost rose to 13.7882. In the individual lines of the table visual, the Cost is calculated by multiplying Quantity by either of those two values (e.g., $27 \times 12.0278 = 324.7506$). In contrast, the value in column Cost (Product Version) is calculated as the individual Quantity times 13.7882 in all rows; this is the standard cost valid for the selected version of the product (e.g., $27 \times 13.7882 = 374.7114$).

⁵ See *Slowly Changing Dimensions.pbix* from the book's GitHub repository for the examples in this section.

Product		Product Version		2023-12-31			
Multiple selections		2023		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

StandardCost for years until November 2021
(12.0278)

StandardCost for years since December 2021
(13.8782)

Figure 10-7. A report page that gives the choice of the StandardCost of each product's version, which can be used to calculate the Cost (Product Version)

To implement this, you need the following parts:

Product version

A table containing the versions, where the report user makes their selection. As new versions can be created at any point in time, it's probably 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 long list of those versions, which could be spread very unevenly over time, making scrolling down the list a bit awkward. But it's totally up to you if you want to exchange the reference to the Date table with `DISTINCT('Product' [EndDate])`.

Resist creating 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 that isn't a StartDate or EndDate. Therefore, we will apply the filter over DAX in the measure where we are calculating Cost (Product Version).

Selected product version

A measure for the selected product version:

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

Standard cost for the selected product version

A measure to find the standard cost for the selected version, independent of the selected date. All the versions of the same product have the identical business key (ProductAlternateKey). Therefore, you need to remove any filters on the product table (a filter on, e.g., the name would be problematic, if the name changes 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 Start Date and EndDate. We need also take into account that StartDate or EndDate could be empty, as the product's version might have always been valid, or might still be 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] = AlternateKey,
            ProductVersion >= 'Product'[StartDate] ||
                ISBLANK('Product'[StartDate]),
            ProductVersion <= 'Product'[EndDate] ||
                ISBLANK('Product'[EndDate])
        )
    RETURN Result
)

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

Hierarchies

If you've followed all the best practices described in this book so far, then you already have denormalized all natural hierarchies in the dimension tables (see “**Denormalizing**” on page 190). With the natural hierarchy denormalized, you have all levels of the hierarchy represented as columns in one table. Adding these columns to a hierarchy is very easy.

Here, let's concentrate on parent-child hierarchies. They're very common, and you also need to store the names of all parents in dedicated columns. Read on to learn how you can achieve this with DAX.⁶

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

```
'Employee (DAX)'[Path] = PATH('Employee (DAX)'[EmployeeKey], 'Employee (DAX)'
[ParentEmployeeKey])
```

Then, you need to dissect the Path and create a calculated column per (expected) level. 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 a variable. Then, you can just copy and 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 n^{th} 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], LevelNumber, INTEGER )
VAR LevelName = LOOKUPVALUE ( 'Employee (DAX)'[FullName], 'Employee (DAX)'
[EmployeeKey], LevelKey )
RETURN LevelName
```

You can already add all `Level` columns to a common hierarchy if you want. I created a matrix visual, shown in [Figure 10-8](#), with the hierarchy on the rows and the measure `Sales Amount` in the value section. So far so good. As soon as you start expanding the upper levels, you'll 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 level shown. The result for the last available level is repeated for all sublevels when they do not have their "own" values.

A good data model should take care of this problem. You need to add another column (to calculate the level an employee is on), a measure to aggregate this column with `MAX`, another measure to calculate on which level the measure is actually displayed, and you also must 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 that of the employee). The unnecessary level won't be displayed if all measures only return blank.

⁶ And see [Hierarchies.pbix](#) to follow along.

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
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

Figure 10-8. The hierarchy expands to unnecessary levels with empty names and repeating Sales Amounts

You can calculate the level of an employee by counting the levels the path contains (by basically counting the separator character plus one). This gives you 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 the function `PATHLENGTH`. Calculating the maximum as a measure is then no real challenge, I guess:

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

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

You need also to determine the level of the measure. Here, you need to check if the column representing a certain level is in the current scope of the visual. If it is, `ISINSCOPE` will return `TRUE`, which is implicitly converted to 1 in an arithmetic calculation. If it isn't in scope, then `ISINSCOPE` will return `FALSE`, which is implicitly converted to 0 in an arithmetic calculation:

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

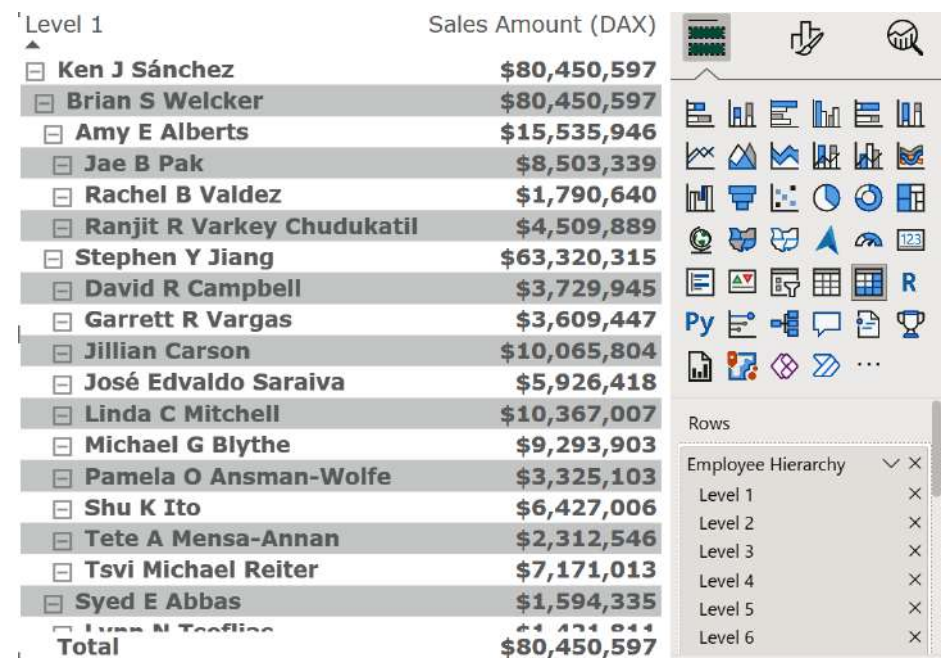
```
ISINSCOPE('Employee (DAX)'[Level 5]) +
ISINSCOPE('Employee (DAX)'[Level 6]) +
ISINSCOPE('Employee (DAX)'[Level 7])
```

If you add columns for additional levels, remember to add them in the calculation of this measure as well.

Finally, add a measure in which you decide whether a value has to be displayed:

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

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 10-9](#).



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
Lyndi K Teefliog	\$1,121,811
Total	\$80,450,597

Rows

- Employee Hierarchy
 - Level 1
 - Level 2
 - Level 3
 - Level 4
 - Level 5
 - Level 6

Figure 10-9. The hierarchy no longer has unnecessary empty levels



Again, calculation groups can be of help, now with hierarchies. You can create two calculation groups items. One to just return the plain measure (`SELECTEDMEASURE()`), and another where you copy and paste the code from measure `Sales Amount` (DAX) and replace `[Sales Amount]` with `SELECTEDMEASURE()`.

Key Takeaways

In this chapter, you learned that DAX is a very powerful language in which you can do every transformation. It is especially useful when it comes to semi- and non-additive calculations, as they can't be implemented outside of DAX (neither with Power Query nor in the data source). Take a look at what you learned in this chapter:

- Normalizing your fact tables involves taking steps to find candidates for dimensions, creating dimension tables as calculated tables via `SELECTCOLUMNS`, and hiding those columns in the fact table. Unfortunately, we can't actually remove those columns from the fact table because it would break the DAX code of the dimension tables.
- Denormalizing in DAX means using `RELATED` to move columns over to the main dimension. Again, we can't remove the referenced tables without breaking the DAX code. Therefore, we just hide these tables.
- I recommend creating all calculations as DAX measures as a starting point (and *not* as DAX calculated columns or as columns in Power Query or SQL). Carefully analyze the formula if it involves multiplication; if it does, you may need to use an iterator function to achieve the correct result.
- You can solve the problem of role-playing dimensions in two ways. You can add (inactive) relationships to the data model and activate them via `USERELATIONSHIP` in the measures where you don't want to use the active relationship. Or add the dimension several times under different names and create standard relationships. Then, no special treatment of your DAX code is necessary.
- Natural hierarchies will have already been denormalized 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.

Now it's time to learn about more advanced challenges, derived from real-world use cases, and how to solve them in DAX.

Real-World Examples Using DAX

DAX and the data model go hand-in-hand. You can solve some challenging use cases directly in the model, so there's no need to write a single line of a DAX formula (this applies to most one-to-many relationships—but there are exceptions, as you will learn in this chapter). Other solutions involve a cooperation between the data model and DAX (like activating an inactive relationship). In rare cases, tables have no relationship defined at all in the data model, but you create a relationship with the `TREATAS` function in DAX, which will only exist during evaluation of the DAX expression.

This chapter covers the following use cases in DAX:

- How DAX can help you to implement *binning* based on a table that defines the ranges of each bucket. Binning is the idea of not showing the actual values but the bucket a value falls into (like small, medium, or large).
- I use a *budget* as an example of a data model with more than one fact table. Here, you need to overcome the challenge that the granularity of one of the fact tables might not be on the primary key level of a dimension table. This leads to a many-to-many cardinality between the fact and dimensions table. DAX can help here.
- No single button in Power BI Desktop creates *multi-language reports*; you need to work around several problems. I prefer a workaround that's solely data-driven and doesn't need any changes in the data model if new translations or new languages are added. To achieve this, I need to add logic to all of my measures.
- *Key-value pair tables* don't have attributes stored in dedicated columns but as individual rows. To make such a table useful for typical reports, you need to *pivot* the table (transforming the information from rows per attribute into columns per attribute).

Binning

In [Chapter 3](#), I introduced you to three different solutions to assign values into bins, and there, I recommended only two: creating a lookup table with a row per each distinct value you want to bin or creating a lookup table containing one row per bin and its lower and upper range value. You can use the *Binning.pbix* file to follow along with the examples in this section.

Lookup Table

To create the lookup table with the distinct values, you can create a *calculated table*. The following code starts with a variable, `_Bins`, containing the distinct values of the `Sales` table's quantity and adds a column containing the bin (with the help of `SWITCH` to check for the given quantity). The second variable, `_BinsWithSortOrder`, builds on the previous one and adds another column, using the minimum quantity for a given bin as the `_SortOrder` column:

```
Bin Table (DAX) = /* calculated table */
VAR _Bins =
    ADDCOLUMNS(
        DISTINCT('Sales'[Quantity]),
        "Bin",
        SWITCH(
            TRUE(),
            Sales[Quantity] < 3, "Low",
            Sales[Quantity] < 5, "Middle",
            "High"
        )
    )
VAR _BinsWithSortOrder=
    ADDCOLUMNS(
        _Bins,
        "_SortOrder",
        VAR _CurrentBin = [Bin]
        RETURN
        MINX(
            FILTER(
                _Bins,
                [Bin]=_CurrentBin
            ),
            [Quantity]
        )
    )
RETURN
    _BinsWithSortOrder
```

You can see the content of the resulting table in [Table 11-1](#). The content of the table appears unordered. This is because the formula derives the content from the `Sales` table; therefore, the rows in the `Bin Table` are in the same order as in the `Sales` table.

Table 11-1. Bin table (DAX)

Quantity	Bin	_SortOrder
3	Middle	3
1	Low	1
4	Middle	3
5	High	5

Don't forget to sort column Bin by column _SortOrder. You can easily achieve this by selecting the Bin column, the fields list of Power BI Desktop, choosing "Column tools" → "Sort by column," and selecting the _SortOrder column, as shown in Figure 11-1.

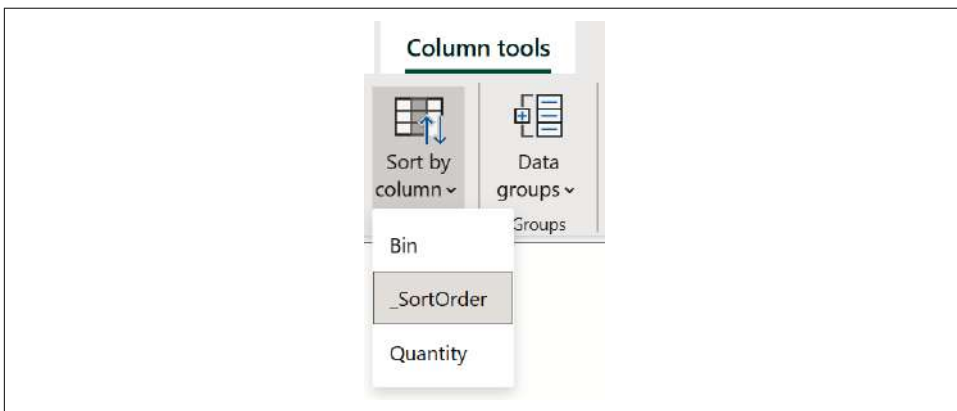


Figure 11-1. Sort the Bin column by the _SortOrder column

Range Table

The other solution, which is discussed in “[Binning](#)” on page 58, is implemented via non-equi-joins from the fact table, which contains the value to be binned to the lookup table, which contains the lower and the upper value of the bin. This table can also be created as a calculated table in DAX.

The first set of parameters of function DATATABLE expects pairs of column names and their datatypes. The second set of parameters must be the data for this table. As in “[Tables](#)” on page 176, you have several options to specify a table in the DAX language. I prefer the {} operator here for brevity. The outer {} defines the whole table. The inner {}s define the content for the individual (three) rows. Here, you don't need an explicit column to order the Bin column. You can just use the existing column Low (incl.) (or High (excl.), for that matter) to achieve this instead.

Here's the code to create the Bin Range table:

```
Bin Range (DAX) = /* calculated table */
DATATABLE(
    // Schema definition
    "Bin", STRING,
    "Low (incl.)", INTEGER,
    "High (excl.)", INTEGER,
    // Rows of data
    {
        {"Low", BLANK(), 3},
        {"Medium", 3, 5},
        {"High", 5, BLANK()}
    }
)
```

The result of the DAX formula is shown in [Table 11-2](#). Pay attention to the blank entries for column Low (incl.) in row Low and for column High (excl.) in row High (created by the BLANK() function in the code).

Table 11-2. Bin Range (DAX)

Bin	Low (incl.)	High (excl.)
Low		3
Medium	3	5
High	5	

The data model in Power BI doesn't allow you to define non-equi-joins, but in DAX we can implement them. The following is important to point out here:

- The bin for the lowest value has an empty Low value. This indicates that we don't care how low the value might be; as long as the value is lower than the specified High value, it will fall into the first bin.
- Similarly, the bin for the highest value has an empty High value. This indicates that we don't care how high the value might be; as long as the value is greater than or equal to the specified Low value, it will fall into the last bin.
- I used the exact same value for the High value of one bin and the Low value of the next bin. This usually makes the table easier to maintain and guarantees that there are no gaps between the bins. Later, you'll implement the lookup so that a value must be greater than *or equal to* the Low value to fall into a bin but lower than the High value.

To make use of the Bin Range table, you need to create a DAX measure that implements the lookup between the 'Sales'[Quantity] and the Bin Range table, like this:

```

[Bin Count] :=
SUMX (
    FILTER(
        'Sales',
        VAR Quantity = 'Sales'[Quantity]
        VAR Bin =
            FILTER(
                'Bin Range',
                NOT ISBLANK(Quantity) &&
                (Quantity >= 'Bin Range'[Low (incl.)] ||
                 ISBLANK('Bin Range'[Low (incl.)] )) &&
                (Quantity < 'Bin Range'[High (excl.)] ||
                 ISBLANK('Bin Range'[High (excl.)]))
            )
        VAR IsInBin = NOT ISEMPTY( Bin )
        RETURN IsInBin
    ),
    1
)

```

The DAX measure consists of three important parts:

A SUMX over the value of 1

Could be easily replaced by a COUNTROWS. I used a SUMX in this example for two reasons: to point out that you could also aggregate facts like quantity, sales amount, etc. instead of just counting rows, and as a reminder that a SUMX(<table>, 1) sometimes performs better than a COUNTROWS(<table>). Always stay flexible in terms of achieving the same result with different DAX formulas, to have something up your sleeve for when performance is not up to expectations.

An outer FILTER

Iterates over the fact table (Sales) and only returns a row if it falls into the bin available in the current filter context.

An inner FILTER

Iterates over the lookup table (Bin Range) and only returns a row when the quantity falls in between the lower and upper range of a bin available in the current filter context. Blank quantities are ignored (NOT ISBLANK(Quantity)), as they fall into none of the bins. The quantity must fulfill the condition of being greater than or equal to the lower bound or lower than the higher bound or the boundary is blank (e.g., ISBLANK('Bin Range'[Low (incl.)])).



If you like the bin range table solution, remember that the non-equi-join implemented in DAX with the two filtered iterations can only be realized in DAX, not directly in the data model or in Power Query or SQL.

Let's turn toward the next use case: multi-fact data models, explained on the example of adding a budget table to a model already containing actual values.

Budget

In “[Budget](#)” on page 60, I lay out that a budget works as a typical example in this book, in which you'll end up with a data model with more than one fact table. And when you create the filter relationship between the Budget table and some of the dimensions, you'll discover that it has a many-to-many relationship, as the relationship is based on neither of the primary keys of the two tables.

One of the three solutions discussed in “[Budget](#)” on page 135 involves DAX and its function `TREATAS`. This function applies a list of values as a filter on a column—and therefore takes over the task you would usually achieve with a filter relationship.



Be aware that creating a relationship in the data model (active or inactive) will create sort of an index on the two columns involved in the filter relationship. The size of the model will increase a bit due to this index but joining the two tables will be sped up. The function `TREATAS` cannot benefit from such an index and is therefore potentially slower. That's why it is important that `TREATAS` shouldn't be your go-to; use it only when a solution with relationships cannot be implemented for good reasons.

In the following code, I use `TREATAS` to manipulate the filter context with the help of `CALCULATE`. I pass in the current filter context's distinct list of `VALUES` of the product's product group as the first parameter and ask to apply this as a filter onto the budget's product group:¹

```
[Budget TREATAS] :=  
CALCULATE (  
    SUM ( Budget[Budget] ),  
    TREATAS (  
        VALUES ( 'Product'[Product Group] ),  
        Budget[Product Group]  
    )  
)
```

I create this measure in a Power BI Desktop file containing the following tables:

- A Date table with a row for every day for the years of data of the fact tables
- A Product table with four rows:

¹ The examples in this section use the [Budget.pbix](#) file on from GitHub.

Product ID	Product desc	Product group
100	A	Group 1
110	B	Group 1
120	C	Group 2
130	D	Group 2

- The Sales table, again with four rows (with sales for three different days), on the granularity level of a single day and a single product:

Date	Product ID	Amount
2023-08-01	100	3,000
2023-08-01	110	20,000
2023-08-02	110	10,000
2023-08-03	120	15,000

- And a Budget table (Table 11-3) with the planned sales amount per day (the first of the month for two different months) and per product group (not per product)

Table 11-3. Budget

Date	Product group	Budget
2023-08-01	Group 2	20,000
2023-08-01	Group 3	7,000
2023-09-01	Group 2	25,000
2023-09-01	Group 3	8,000

The measure Budget TREATAS in Figure 11-2 shows the correct budget per month and product group. The total is calculated as the sum over all product groups available in the Product table (and matches the values displayed in the table visual). Unfortunately, in the example we also have a budget for product groups, which are not available in the Product table (as no products for these groups are defined yet).

Month	Group	Sales Amount	Budget TREATAS
2023-02	Group 1	33000	
2023-02	Group 2	15000	20000
2023-03	Group 2		25000
Total		48000	45000

Figure 11-2. Budget TREATAS per day and product group

If you want to solve the problem of foreign keys in the fact table, which aren't available in the dimension table, then you need a different approach. You need to create a table that contains all product groups, by combining the product groups from both the product and the budget table. No duplicated product groups are allowed in this table. Afterward, you can use this table as a *bridge* table, as described in “Budget” on page 135, to create two one-to-many relationships from this table, to the product table and to the budget table.

Find the DAX code for the bridge table created as a calculated table here:

```
Product Group BRIDGE = /* calculated table */  
DISTINCT (  
    UNION (  
        DISTINCT ( 'Budget'[Product Group] ),  
        DISTINCT ( 'Product'[Product Group] )  
    )  
)
```

The next challenge is to create the necessary logic in DAX to supply the multi-language model.

Multi-Language Model

In this section, you'll use DAX to apply the selected language (row in the Language table) to the (other) dimension tables, using the *Multilanguage.pbix* file. In “Multi-Language Model” on page 139, you learned that Power BI doesn't allow you to create active filter relationships between the language table and more than one dimension table (and complains about an *ambiguous model* instead).

Therefore, you need to find a way to apply the filter for a language within the DAX measure that calculates, e.g., the sales amount. You need to make sure that a non-blank value is only available for dimension rows with the correct language. For the other languages, it should return BLANK. Power BI's default behavior will make sure such blank values aren't displayed, and therefore omit dimension rows in the “wrong” (not selected) language.



As a fail-safe, always make sure to use the function `SELECTEDVALUE` when accessing the selected language to display the language or use the language to filter the dimension table. This function will return the selected value. In case no or multiple selections have been made, you can provide a fallback value (e.g., English). In a perfect world, this wouldn't be necessary; every report creator would make sure to set the slicer properties or filter for the language selection to force single selection. But as a model creator, you have no control over this. It's better to be safe than sorry.

Before I show my preferred solution, I want to contemplate four options:

Use USERRELATIONSHIP

Power BI allows you to create the relationships between the language table and the dimension tables as inactive relationships. An *inactive* relationship is not part of the filter context, unless it is explicitly activated in a measure. Activating is done via function USERRELATIONSHIP, as “Relationships” on page 179 describes:

```
Sales (USERRELATIONSHIP) =  
CALCULATE(  
    SUM('Fact'[Amount]),  
    USERRELATIONSHIP('Language'[Language ID], Dim1[Language ID])  
)
```

This works very well when you only activate a single non-active relationship, because DAX will automatically de-activate the active relationship (to avoid more than a single relationship being active). Unfortunately, this does not work at all in the use case of the Language table and relationships to two dimension tables, as the proper solution needs to activate more than one of the inactive relationships (all relationships between the Language table and the dimension tables), to propagate the filter from the Language table to the dimension tables:

```
Sales (USERRELATIONSHIP) =  
CALCULATE(  
    SUM('Fact'[Amount]),  
    USERRELATIONSHIP('Language'[Language ID], Dim1[Language ID]),  
    USERRELATIONSHIP('Language'[Language ID], Dim2[Language ID])  
)
```

This formula leads to the confusing error message: *USERRELATIONSHIP function can only use the two columns references participating in relationship*, which just means that you can’t activate both inactive relationships. This idea can’t be implemented successfully.

Use USERCULTURE

Another solution could ignore the filter on the language table for the case of displaying the dimension’s names, and therefore avoid all problems of ambiguous relationships in the data model. You could use the USERCULTURE function to access the user’s language. USERCULTURE returns a string in the format <language>-<CULTURE>. For example, en-US for the language English and US/American formatting. (Make sure to use lowercase and uppercase as in the example, as this string is case sensitive.)

You can set the preferred language in Power BI Desktop (File → “Options and settings” → Options → GLOBAL → Regional Settings) for reports opened in Power BI Desktop and in the browser’s settings in the case of the Power BI service. You can always explicitly override the browser’s settings by adding a

parameter to the URL. For example, add `?language=en-US` to set the language explicitly to English and all formatting options to US.

The following example uses `TREATAS` to set the dimension's Language ID obtained from these settings. In the DAX formula, I first set the content of variable `LanguageID` to the first two characters of the result of `USERCULTURE` (which represents the language). I wrap the expression into `{}` to convert the scalar value into a table (which contains only one column and one row), as `TREATAS` expects a table expression as its first parameter. As in the solution with `USERRELATIONSHIP`, I then use `CALCULATE` to change the filter context for the expression; this time I apply the variable `LanguageID` as filters to the dimension's Language ID columns:

```
[Sales (USERCULTURE)] =  
VAR _LanguageID = {LEFT(USERCULTURE(), 2)}  
RETURN  
CALCULATE(  
    SUM('Fact'[Amount])  
    ,TREATAS(_LanguageID, Dim1[Language ID])  
    ,TREATAS(_LanguageID, Dim2[Language ID])  
)
```

This can be beneficial when you want to tie the display language to the browser's settings (instead of offering a slicer to change the language ad hoc in the report). The drawback is that you need to apply the logic in `CALCULATE` to each and every measure. To make maintenance easy, you could create one calculation group, which takes care of the filtering, as shown here:

```
VAR _LanguageID = {LEFT(USERCULTURE(), 2)}  
RETURN  
CALCULATE(  
    SELECTEDMEASURE()  
    ,TREATAS(_LanguageID, Dim1[Language ID])  
    ,TREATAS(_LanguageID, Dim2[Language ID])  
)
```

Create one slicer per dimension on the dimension's language key

Another approach to ensure all dimensions are filtered by the selected language is to create one slicer per dimension on the dimension's language key (not on the Language table). These slicers can be synchronized with each other, by choosing View → “Sync slicers” → “Advanced options” from the ribbon. Make sure to input the same string for all to-be-synced slicers under “Enter a group name to sync selection to any other visuals with that group name” (see [Figure 11-3](#)).

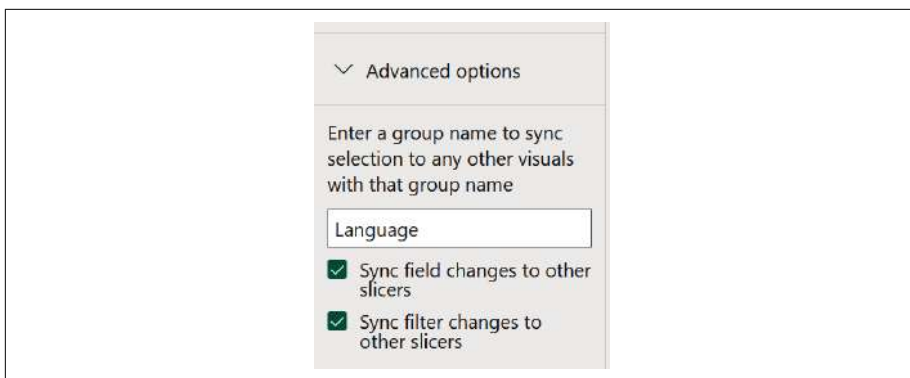


Figure 11-3. Advanced options allows to keep slicers on different fields in sync

There are drawbacks, though. First, filters on the lookup table (Language) might be more performant than filters created on the foreign keys (the Language ID columns of the dimension tables). Second, the content of foreign key might not be user-friendly (e.g., “en” instead of “English”)—so you need to store the full language’s name in every dimension if you want to keep the slicers content easy to use. Finally, and I think most importantly, this solution must be implemented in the report tool (and is not covered by the data model). If somebody else is using your report’s dataset as a data source, they need to reimplement the synchronization of the slicers (to not suffer from duplicated dimension rows and duplicated values). Some tools (like Excel) may not allow synchronizing filters.

Statically assign users to roles or use dynamic row-level security

The problem with propagating the language’s filter on several dimensions can also be solved via row-level security. A filter based on row-level security is activated on a different layer, so to speak, than the usual filter context. If a security role filters multiple tables with the same filter, you do not receive the “ambiguous model” error message (as in the case when I tried to have several active relationships from the Language table to the dimension tables). You can either statically assign users to a role matching their preferred language or you can implement dynamic row-level security through a lookup table that provides the user’s *universal principal name* and the language. I implemented the latter in [Table 11-4](#).

Table 11-4. A table containing users and their preferred languages

User name	Language ID
James@savorydata.com	EN
Fritz@savorydata.com	DE
Jens@savorydata.com	DA
Koloth@savorydata.com	tlh-Latn

First, you need the lookup table, which provides the security context per user. For example, user `James@savorydata.com` has Language ID EN assigned.

Make sure that an active filter relationship between the Users and Language tables is created on the basis of the Language ID column. The relationship must have a one-to-many cardinality (with table Languages on the “one” side). In this example, Power BI will suggest a one-to-one cardinality because the Users table only contains one row per language. But in reality, of course, there could be many rows for the same language in Users; multiple users will want to consume the report in the same language. As a filter direction, choose *Both*, as you want the Users table to filter the Language table (and not the other way around), as shown in [Figure 11-4](#).

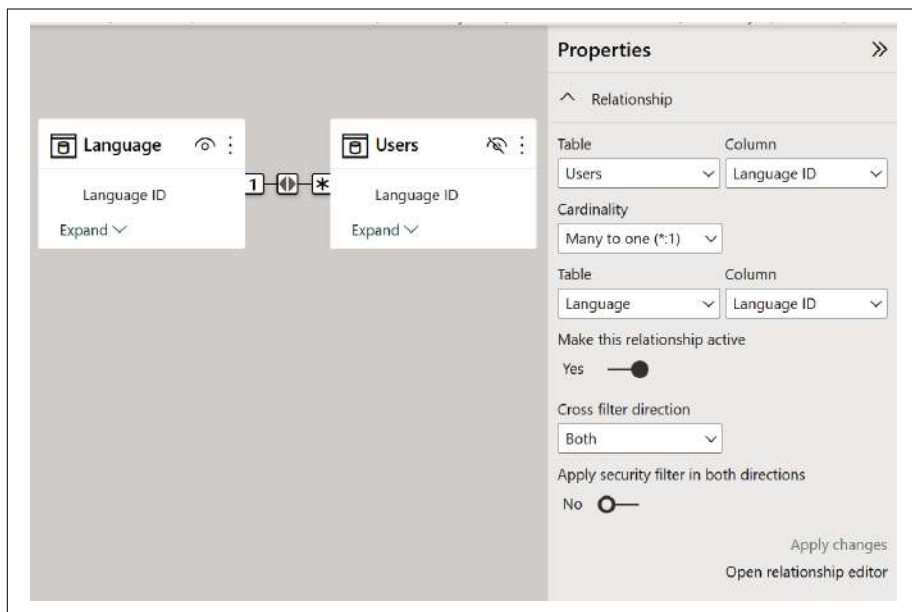


Figure 11-4. One-to-many bi-directional relationship between tables Language and Users

Then choose Modeling → “Manage roles” → Create to create a new role. I name the role USERNAME because it facilitates the USERNAME function to create the right filters. In this role, you must create a filter for every dimension table (e.g., Dim1 and Dim2) where you let Power BI look up the Language ID column from the Users table, where the User Name column contains the name of the currently signed-in user (returned by function USERNAME).

```
[Language ID] =  
    LOOKUPVALUE('Users'[Language ID], 'Users'[User Name],  
    USERNAME())
```

I recommend creating a filter on the Users table as well:

```
[User Name] = USERNAME()
```

Test to see if the role assignment works by clicking Modeling → “View as,” selecting first “Other user,” and entering the user for whom you want to simulate the experience. Then, select the newly created role (USERNAME, in this case). **Figure 11-5** shows the dialog box in the foreground and the resulting filtered report in the background.

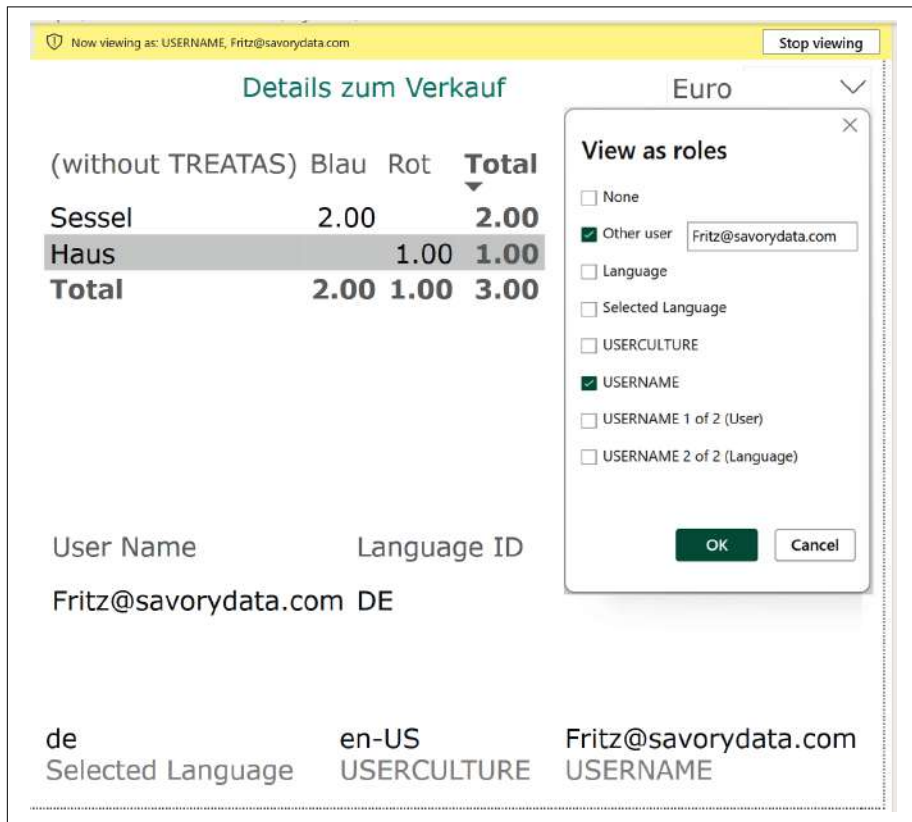


Figure 11-5. User *Fritz@savorydata.com* will be presented with the German version of the report

However, I see drawbacks to using row-level security to filter for a language. First, the user can't change the language dynamically, but needs to either have write permission on the Users table (to change the content of the Language ID for their entry) or ask someone to change their role assignment there.

Row-level security works only for users with a *Reader* role in the Power BI service. If you're a member, contributor, or admin in the workspace (or the admin of the Analysis Services database), any other role assignment is ignored. Therefore, no filter on the language will be in place for you, and you'll see instead all the values for all dimensions multiple times, for each existing language. Every report author would need a second read-only user so he can develop and test reports. To tell you the truth, I personally find using row-level security as a workaround for language selection to be like using a sledgehammer to crack a nut.

For myself, I'd prefer to let the user decide on the display language. Instead of USER CULTURE, you can, with the exact same effort, also apply the selected language (of the Language table) to the dimension tables. The only difference is the expression for the _LanguageID variable. It uses SELECTEDVALUE to return the selected Language ID. If no selection is made, or more than one value is selected, the function will return the content of the optional second parameter (EN in this case):

```
[Sales] :=
VAR _LanguageID = {SELECTEDVALUE('Language'[Language ID], "EN")}
RETURN
CALCULATE(
    SUM('Fact'[Amount])
    ,TREATAS(_LanguageID, Dim1[Language ID])
    ,TREATAS(_LanguageID, Dim2[Language ID])
)
```

Again, you need to apply this to each and every measure—or create a calculation group, as shown here:

```
VAR _LanguageID = {SELECTEDVALUE('Language'[Language ID], "EN")}
RETURN
CALCULATE(
    SELECTEDMEASURE()
    ,TREATAS(_LanguageID, Dim1[Language ID])
    ,TREATAS(_LanguageID, Dim2[Language ID])
)
```

In an ideal world, you could create a filter that the user can change interactively, but can be applied to all (dimension) tables, similarly to how row-level security works, and without the annoying hint about the ambiguity of the data model. I proposed this as an idea a couple of years ago. You can vote it up in the [Microsoft Fabric Community](#).

Now it's time to face the next challenge: pivoting key-value pair tables with the help of DAX.

Key-Value Pair Tables

Key-value pair tables have attributes stored row-by-row (see “[Key-Value Pair Tables](#)” on page 65). Many reporting tools (Power BI included) benefit when such a table is pivoted and each key-value pair (row) is transformed into an explicit column. For this solution, you need to first distinguish between *aggregable* columns (typically numeric) for which you want to calculate and show a total (e.g., a customer’s revenue), and those for which showing information in a totaled form doesn’t make sense (e.g., a customer’s city of residence).

Let’s start with the nonaggregable columns.² The very inner part of the following measure calculates the minimum value for column `Value`. In the context of a measure, you always need to apply an aggregate function when referencing a table’s column. Practically, there will be only one value available at the point in time when `MIN` is executed because it’s embedded into an `IF` with a condition that only one `ID` is available in the current filter context (`HASONEVALUE`). This safeguard keeps the total (a filter context where more than a single `ID` is available) blank. Theoretically, there could be multiple cities for one `ID`, and then you’d select the (alphabetically) first value. The whole expression is wrapped inside a `CALCULATE`, which manipulates the filter context to only a `Key` of `city`. That’s because the measure should only return the value for a city:

```
[City] :=  
CALCULATE(  
    IF(  
        HASONEVALUE('Source'[ID]),  
        MIN('Source'[Value])  
    ),  
    KEEPFILTERS('Source'[Key] = "city")  
)
```

For aggregable values, you use one of the fitting iterator functions (in this example, it’s `SUMX`) to iterate over the key-value pair table, where the `Key` is “revenue” (function `FILTER`) and aggregate the content of `Value`. As the column `Value` is of a “string” data type (so it can contain data of any data type), I use the `VALUE` function to convert its content explicitly to a numeric value:

```
[Revenue] :=  
SUMX(  
    FILTER(  
        'Source',  
        [Key] = "revenue"  
    ),  
    VALUE([Value])  
)
```

² This section’s examples use the *Key Value.pbix* file on GitHub.

```

        VALUE([Value])
    )

```

In this static approach, you need to take one or the other pattern (aggregable or non-aggregable columns) and create measures for every single key of the key-value pair table. By now, you might know that I'm not a huge fan of such static solutions. I don't think it's a good idea to have a solution in place where you need to involve a developer every time data changes.

New keys can be added at any time to the key-value pair table. That's why I developed the following generic measure. It contains two variables, one for each approach (NumericValue for the aggregable cases, NonNumericValue when the column should be treated as a string). For NonNumericValue, I remove the CALCULATE and filter; for NumericValue, I still need the filter to avoid error messages indicating that some Value (e.g., "Seattle") could not be successfully converted to a numeric value.

At the very beginning of the code, I create another variable (NumericColumns), which is a table containing all those keys that should be treated as numeric (aggregable). This table variable is used in variable NumericValue (instead of the static value discussed in the static solution) and in variable GenericValue to decide which of the two variables (NumericValue or NonNumericValue) should be returned as the result of this measure:

```

[Generic Value] :=
VAR NumericColumns = {"revenue"}
VAR NumericValue =
    SUMX(
        FILTER(
            'Source',
            [Key] IN NumericColumns
        ),
        VALUE([Value])
    )
VAR NonNumericValue =
    IF(
        HASONEVALUE('Source'[ID]),
        MIN('Source'[Value])
    )
VAR GenericValue =
    IF (
        ISBLANK(NumericValue),
        NonNumericValue,
        NumericValue
    )
RETURN GenericValue

```

Instead of creating a new measure for every key, you only need to maintain the list of numeric columns within one measure. What an improvement in maintenance!

If you're as lucky as I am with the given example, the key-value pair table contains a column to specify the key's data type. In the following code example, I trust that the names of Power Query's numeric data types are used to describe the actual data type of a key's value ("Number", "Currency", "Percentage", "Int64.Type"). But you can easily change the code to look for 1 or for integer or any value that the key-value pair table's creator used to identify numeric keys:

```
[Generic Value (type)] :=
VAR NumericColumns =
DISTINCT(
    SELECTCOLUMNS(
        FILTER('Source', 'Source'[Type] IN
            {"Number", "Currency", "Percentage", "Int64.Type"}),
            "Key", [Key]
        )
    )
VAR NumericValue =
SUMX(
    FILTER(
        'Source',
        [Key] IN NumericColumns
    ),
    VALUE([Value])
)
VAR NonNumericValue =
IF(
    HASONEVALUE('Source'[ID]),
    MIN('Source'[Value])
)
VAR GenericValue =
IF (
    ISBLANK(NumericValue),
    NonNumericValue,
    NumericValue
)
RETURN GenericValue
```

With this version of the generic measure, you don't need to change anything when new data arrives, as long as the Type column delivers the correct content. Maintenance went down to zero.

As a bonus, here's another version of the generic measure. The following code concatenates the content of NonNumericValues instead of picking the alphabetical first row:

```
[Generic Value (type, concatenated)] :=
VAR NumericColumns =
DISTINCT(
    SELECTCOLUMNS(
        FILTER('Source', 'Source'[Type] IN
            {"Number", "Currency", "Percentage", "Int64.Type"}),
            "Key", [Key]
        )
    )
VAR NumericValue =
SUMX(
    FILTER(
        'Source',
        [Key] IN NumericColumns
    ),
    VALUE([Value])
)
VAR NonNumericValue =
CONCATENATEX(
    FILTER(
        'Source',
        [Key] NOT IN NumericColumns
    ),
    [Value],
    ""
)
VAR GenericValue =
IF (
    ISBLANK(NumericValue),
    NonNumericValue,
    NumericValue
)
RETURN GenericValue
```

```

        "Key", [Key]
    )
)
VAR NumericValue =
    SUMX(
        FILTER(
            'Source',
            [Key] IN NumericColumns
        ),
        VALUE([Value])
    )
VAR NonNumericValue =
    IF(
        HASONEVALUE('Source'[ID]),
        CONCATENATEX(
            VALUES('Source'[Value]),
            [Value]
        )
    )
)
VAR GenericValue =
    IF (
        ISBLANK(NumericValue),
        NonNumericValue,
        NumericValue
    )
RETURN GenericValue

```

Combining Self-Service and Enterprise BI

Creating calculated columns and calculated tables is prohibited in live connection mode. If you need to add such artifacts, you need to convert a live connection into DirectQuery mode. Independent of the data source and the storage mode (import, DirectQuery or live connection) you can always add measures written in DAX to the data model.

I strongly recommend that you only store calculations for ad hoc analysis in a local data model. Before you start publishing the report (including the calculation), you should make sure that the calculation is moved into the central (enterprise) data model, so everybody can benefit from it. You only need to copy the DAX formula and paste it into an email to the data engineer responsible for the centralized analytic database.

Key Takeaways

This is the last chapter about modeling data with DAX. You learned how to use DAX in the following use cases:

- Create the lookup tables containing the definitions of the buckets of each value for the binning problem. You can also use DAX to create a measure implementing a non-equi-join if the bins are specified via ranges of values instead.
- Propagate filters with the TREATAS function when you can't or don't want to create a relationship in the Model view. Examples in this chapter covered the case of non-equi-joins, an alternative to implement many-to-many relationships and avoid ambiguous data models.
- Add TREATAS to either every measure or create a calculation group item to propagate the user's language into the dimension tables.
- Apply the appropriate filter to create a DAX measure per Key of a key-value pair table, and thereby pivot the content. If you want to avoid creating (over and over again) a DAX measure per (new) key, you can use the generic approach described in this chapter. A column describing the true data type of the content of the value column is very useful.
- Use DAX in both self-service BI and enterprise BI to enrich the data available. Keep in mind, though, that the best place for standard calculations is always the centralized data store and not the report layer. Therefore, move calculations from a Power BI Desktop file to the centralized data model, as soon as you have verified its correctness.

And remember, when it comes to non-additive and semi-additive calculations, there is no way around explicit DAX measures. For additive calculations, I recommend DAX measures as well. For general transformations of a data model, Power Query and SQL are the better choice. **Part IV** introduces you to the role Power Query plays when you model your data for Power BI.

Performance Tuning with DAX

In this chapter, I show how to create the necessary tables in DAX to support the performance tuning concepts from [Chapter 8](#). The idea is to create additional tables that contain the data in an aggregated way. These additional tables increase your semantic model's memory consumption in exchange for faster reports (less data has to be read to create the result of a query). For simpler aggregation tables, you can configure Power BI to automatically use the aggregation tables. For more complex scenarios, you need to add logic to your measures so they use the detailed or the aggregated table instead for their calculations. You can use the *Performance Tuning.pbix* file to follow along with the examples in this chapter.

Storage Mode

Analogously to calculated columns, calculated tables are always persisted in the data model and re-calculated during the refresh of the data model. With that said, calculated tables (written in DAX) are always in Import mode—independently of whether the source of the table expression is in DirectQuery or Import mode. You cannot add calculated tables in live connection mode.

Pre-Aggregating

You can group by one or more dimension columns and create aggregations on this aggregation level with the `SUMMARIZECOLUMNS` function's help. In the first parameter(s), you provide column names for the dimensional values you want to group on.

For the combination of the values of these columns, the aggregation table is generated. These columns specify the granularity of your aggregation table. Then you provide pairs of parameters. The first of these pairs is a string and specifies the name of

the resulting column. The second member of this pair is a DAX expression (to calculate the aggregated value).



Keep in mind that SUMMARIZECOLUMNS iterates over the virtual table generated from the first parameters. Therefore, the DAX expression for the value is calculated in a row context. Make sure to wrap the expression in CALCULATE when you need a filter context instead.

The table generated from the following code is aggregated on only the OrderDate column. For every available OrderDate of the Reseller Sales (DirectQuery - Agg) table, the Sales Count (number of transactions), and the Sales Amount are calculated:

```
Reseller Sales (Agg Table DAX) = /* calculated table */
SUMMARIZECOLUMNS(
    'Reseller Sales (DirectQuery - Agg)'[OrderDate],
    "Sales Count", CALCULATE(COUNT('Reseller Sales (DirectQuery - Agg)'
        [SalesAmount])),
    "Sales Amount", CALCULATE(SUM('Reseller Sales (DirectQuery - Agg)'
        [SalesAmount]))
)
```

Reports requesting any of the two numbers calculated per day can be directly satisfied (as a single row) from this table. Reports requesting these numbers on, e.g., a monthly basis can also be satisfied; only about 30 rows would have to be aggregated.

Queries that ask for numbers per product (or day and product) can't be satisfied from this table. You need to make sure that either Power BI is aware of how to use this aggregation table (as explained in [“Pre-Aggregating” on page 166](#)) or that you add logic to your measures accordingly. The latter is explained in the next section.

Aggregation-Aware Measures

In cases where the out-of-the box features of Power BI's “Manage aggregations” don't match all the necessary requirements (see [Chapter 8](#)), you can make any measure *aggregation aware* by adding a condition to return an expression based either on the original transaction table or on the aggregation table. The task of the condition is to find out if the current filter context contains only dimension tables referenced by the aggregation table. If this is the case, then the calculation can be safely based on the aggregation table. If the current filter context also contains other dimension tables, you need to base the calculation on the transaction table. Shoutout to Marco Russo, who helped me to write this condition in a way where I list the dimension tables used in the aggregation table (instead of checking all the other dimensions).

The aggregation table `Reseller Sales (Agg Table DAX)`, created in the previous section, is aggregated on the `Date` level. Therefore, this table can be used to satisfy calculations that have no filter at all or filter on the date dimension. It can't be used if a filter on the `Product` table is present in the current filter context. In this case, the transaction table `Reseller Sales (DirectQuery - Agg)` has to be used. You see an example of such an expression in the following code:

```
[Sales Amount] :=
IF(
    CALCULATE ( NOT ISCROSSFILTERED ( 'Reseller Sales (DirectQuery)' ),
        REMOVEFILTERS ( 'Date (Dual)' ) ), // Kudos to Marco Russo
    [Sales Amount (Agg Table)],
    [Sales Amount (DirectQuery)]
)
```

The first parameter of the `IF` function is a condition to check if the (non-aggregated) transaction table is (still) cross-filtered after the filter on the `Date (Dual)` table is removed. If there are other filters in the current context, then this condition returns `FALSE` and the expression needs to return the sales amount calculated over the transaction table (`[Sales Amount (DirectQuery)]`). If the transaction table is only filtered by the `Date (Dual)` table or is not filtered at all, the expression can safely calculate the sales amount from the aggregated table (`[Sales Amount (Agg Table)]`).

You could also write the `IF` condition the other way around: not checking for filters on the dimensions the aggregation table is based on, but on the dimensions the aggregation table is *not* based on. If `ISCROSSFILTERED('Product')` is true, then the sales amount cannot be safely calculated from the aggregation table, but must be calculated from the transaction table.

The problem I see, though, is that if you add another dimension to the data model at a later point in time (referenced by the transaction table), then you'll need to change the measure and add `ISCROSSFILTERED(<New Dimension>)` because you need to continue to check for filters on dimensions the aggregation table is *not* based on. There's probably a chance that you'll overlook a measure and then end up with incorrect calculations. Therefore, I prefer checking on the dimensions used in the aggregation table instead.

Key Takeaways

The performance tuning concept in the data model is based on exchanging storage for query runtime, when query runtime is longer than report users would like. DAX can support this concept in two ways: one, you can create aggregated tables as calculated tables, and two, you can make any measure aware of such aggregations to perform a calculation in the most efficient way.

You also learned the following in this chapter:

- Calculated tables are always in Import mode (independent of the mode of the base table).
- Table expressions in DAX allow you to create calculated tables. They can have any content. You can create tables with aggregated content from another table with the help of the SUMMARIZECOLUMNS function.
- If the capabilities of the “Manage aggregations” feature aren’t enough, you can implement logic of any complexity in a measure via IF or SWITCH. You decide if the calculation should be done based on the transaction table or based on any of the pre-aggregated tables.

This chapter concludes the part about DAX. As you’ve learned, DAX is very powerful in creating calculated columns and calculated tables in a data model. These artifacts are always persisted in the data model and occupy memory. If you want to aggregate tables in DirectQuery mode, you can do so with the help of Power Query and SQL. **Part IV** kicks off with Power Query.

Data Modeling for Power BI with the Help of Power Query

Chapters	Parts				
	Part 1 Data modeling in general	Part 2 Power BI	Part 3 DAX	Part 4 Power Query	Part 5 SQL
Understanding a data model	Chapter 1	Chapter 5	Chapter 9	Chapter 13	Chapter 17
Building a data model	Chapter 2	Chapter 6	Chapter 10	Chapter 14	Chapter 18
Real-world examples	Chapter 3	Chapter 7	Chapter 11	Chapter 15	Chapter 19
Performance tuning	Chapter 4	Chapter 8	Chapter 12	Chapter 16	Chapter 20

This part of the book is all about Power Query. Power Query's purpose is to bring the information available in any data source into the right shape. Even when the data is already in the correct shape, it runs "through" Power Query. You will come very far by clicking the correct buttons in Power Query's user interface. Every transformation you are applying to the data is "recorded" as a step in a Power Query query, written in the M language. Only in more advanced cases, though, will you need to lay your hands on such a script.

This part starts with an overview about the basic terms and concepts (in [Chapter 13](#)):

- Tables and queries
- Merging columns to form a primary key
- Creating a surrogate key
- Combining queries

[Chapter 14](#) will show you the steps you'll probably need to apply to all your data sources:

- Normalizing and denormalizing
- Adding calculations
- Transforming flags and indicators into meaningful text
- Adding a date or time table
- Duplicating dimension tables per role
- Treating slowly changing dimensions
- Flattening parent-child hierarchies

This part teaches you how to solve the real-world examples introduced in [Chapter 3](#) with the help of Power Query ([Chapter 15](#)):

- Binning
- Multi-fact data models
- Multi-language data models
- Key-value pair tables

Power Query has some interesting features that allow you to optimize and find the best trade-offs among the available storage modes to increase the speed of reports and queries based on a data model. [Chapter 16](#) has you covered here.

Understanding a Data Model from the Power Query Point of View

Power BI's tool to create the tables for and maintain the shape of the data model is Power Query. Power Query is directly built into the product and can be accessed via Home → “Transform data.” It opens in a separate window, which is convenient if you have more than one screen. Then you can make changes to the transformations in Power Query, refresh the data model, and test in the reports without closing the Power Query window.

All changes you make in the UI in the Power Query window are “recorded” as steps, which are applied to the source data. You can re-access every step (which is convenient when you need to debug the transformations). For many of the steps, you can also click on the gear icon to change a step through a dialog box (which, in most cases, matches the dialog box you use when you create a step in the first place). Throughout **Part IV**, I also show how to directly change the steps or edit a script in the *advanced editor*. The language of the script is the Power Query mashup language, or M for short. Unfortunately, it doesn't have much in common with the DAX language.

Power BI dataflows are sometimes called “Power Query online.” They share most of the functionality of Power Query. Unfortunately, they aren't completely identical in terms of features, but so close that you can copy most M scripts between the two tools. Azure Data Factory offers a similar experience via Data Wrangling in Azure Data Factory. Again, there are different limitations compared to Power Query in Power BI Desktop.

Data Model

All the data landing in Power BI's data model comes via Power Query. There is no way around it. It plays a very important role, but Power Query isn't part of the data model itself.

I prefer implementing transformations in Power Query to implementing them in DAX for several reasons. First, Power Query offers a rich graphical user interface (GUI) where you can achieve even complex tasks without writing a single line of code. This makes creating transformations fast and easy. Every transformation is recorded as an “Applied step.” Many types of steps offer a gear icon (e.g., Pivoted Column, shown in [Figure 13-1](#)), which brings back a dialog box where you can change the properties of the step. You can directly edit the code as well: either you turn on View → Formula Bar and edit the code for a single step in the formula bar, or you use View → Advanced Editor to edit the whole script behind a Power Query query. This is necessary when a feature of the M language hasn't made it into the GUI yet, for example—like some parameters of functions—or if you need to solve more complex algorithms, such as programming a loop to iterate over items or reference intermediate steps of the Power Query code.

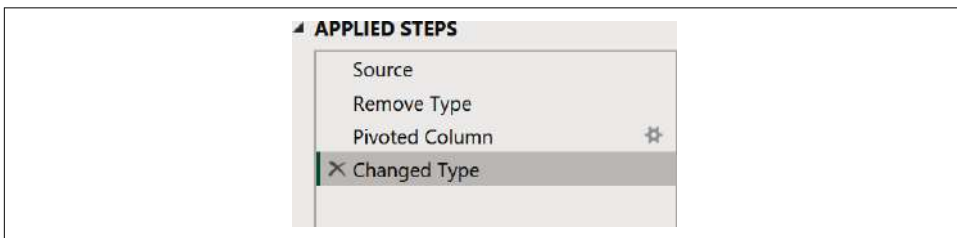


Figure 13-1. Applied steps of a Power Query

Second, in Power Query, you can decide if the result should be loaded into Power BI's data model per query. [Figure 13-2](#) shows that the option “Enable load” doesn't show a checkmark and displays the name of query SQL in italics. Disabling this option is useful when a certain query should not become a table in the Power BI data model. If you were to postpone transformations to DAX instead, you'd need to load the “half-baked” tables and combine them in the correct manner into calculated tables and/or calculated columns to shape the data model. You can hide those half-baked tables, but they'll still take up memory and be updated with every refresh (slowing down your data model's overall refresh time).

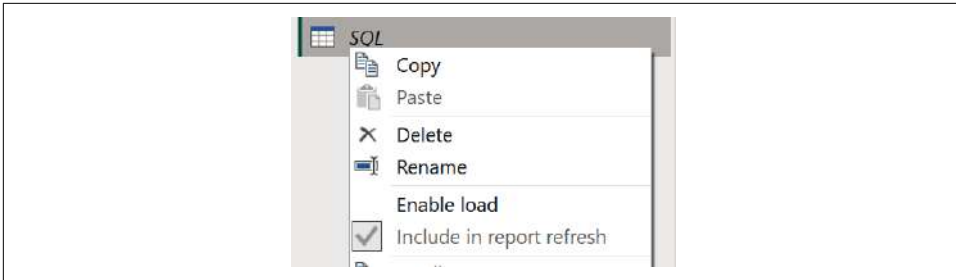


Figure 13-2. Applied steps of a Power Query

For some data sources (see “[Query Folding on Native Queries](#)”), Power Query will translate your transformations into the data source’s query language and, hence, push the hard work to the data source. This saves Power Query (and your machine or resources in the Power BI service) from downloading all the raw data before your transformations are applied step-by-step. You can imagine that a database server is much better at applying filters, selecting columns, and completing transformations than your local hardware will ever be. This feature is called *query folding* and leads to a faster refresh of the data model (and query response time in case of DirectQuery storage mode).



When you plan to enable DirectQuery or Dual mode for a table, you need to make sure that the transformations applied are only so complex that query folding is possible for the table (see [the documentation](#)). Otherwise, the query response time of reports and visuals will not be satisfying.

Lastly, I prefer using Power Query for transformations because compression of the data model tends to be better if you import it through Power Query as opposed to adding calculated columns via DAX. This has something to do with the fact that the order of the rows directly influences the compression ratio for your data. When loading data from Power Query, Power BI tries to find an optimal sort order of the rows of a table. When you later add calculated columns, the sort order of the rows will not be changed—resulting in a potentially less-than-optimal sort order and, therefore, less-than-optimal overall compression of the column’s data.

The drawback of adding columns via Power Query is that you need to refresh the whole query/table afterward. When you add or change the definition of a calculated column in DAX, the new column is calculated based on the already imported data—no need for any data refresh. For a quick proof-of-concept, a DAX calculated column can be the better choice. But I recommend that you move the definition to Power Query before you make the model available to other people.

Tables are the basis of every data model. Learn how Power Query queries are related to Power BI's tables in the next section.

Basic Components

A data model consists of the following parts, which Power Query can provide:

- Tables or queries
- Relationships
- Primary keys

Tables or Queries

Power Query creates and maintains queries (hence the name Power Query), not tables. But it is important to know that a query created in Power Query ultimately becomes a table in the data model (unless you disable the “Enable load” option for a query). Therefore, the result of a Power Query transformation is sometimes also called a table. All transformations (including choosing the data type, names of tables and columns, etc.) are matched into the data model. The collaboration between the two is very strong. If you, for example, change the name of a column in Power BI in the Data pane, you will discover that there is a step, “Renamed columns,” added in Power Query reflecting this change.

The syntax of Power Query/M scripts is unfortunately completely different from DAX. While DAX is similar to Excel's formulas, M is similar to (and based on) the F# language. Most importantly, M is case sensitive (it distinguishes between lower- and uppercase characters in both keywords and string comparisons), while DAX is case insensitive.



The fact that Power Query is case sensitive and the Power BI data model (or DAX) isn't can sometimes bite you. For example, if you remove duplicates in Power Query, it'll identify text like “ABC123,” “Abc123,” and “abc123” in, say, column ProductID in the Product table as different values (and therefore keep all three values).

If you want to use that ProductID column to create a relationship in your data model, the Product table's ProductID column can't be on the “one” side of the relationship. Power BI will force you into a many-to-many relationship between the Product table and the Sales table, for example. The solution here is to transform the content of the ProductID column into uppercase or lowercase before you let Power Query remove duplicates.

I use Power Query/M to prepare the data but will not use it to create any non-additive calculations, as they can only be correctly provided in DAX.

Power BI allows you to define on a query-per-query basis if you want to load the result into the Power BI data model. This allows for queries that contain only intermediate steps, like normalized tables, which you will merge with other tables to achieve a unified dimension table.

In Power Query, a column can be any of the following data types ([Figure 13-3](#)):

Decimal number

Decimal number is stored as a 64-bit floating-point number. Such numbers have a range from $-1.79E +308$ through $-2.23E -308$, 0, and positive values from $2.23E -308$ through $1.79E + 308$, but only with a maximum precision of 15 digits.



Power Query suggests Decimal number when you let it detect the data type. In general, you should avoid floating-point numbers because they can't represent all numbers within their supported range with 100% accuracy (in accordance with the IEEE 754 Standard, as [the documentation](#) points out). Only use them if Fixed decimal number won't work for you.

Fixed decimal number

Despite the \$-icon suggesting this data type is built for currency amounts, a Fixed decimal number is made for all types of numbers, as long as a precision of 19 digits, from which 4 fixed digits are reserved for the decimals, is good enough (which has been the case for most of the data models I've built so far). This data type can cover numbers ranging from $-922,337,203,685,477.5807$ through $+922,337,203,685,477.5807$. Fixed decimal number should be your default choice for all numbers which are not whole numbers, as full accuracy is guaranteed.

Whole number

Whole number is identical to Fixed decimal number with one exception: no digits are reserved for the decimals. Whole number covers numbers ranging from $-9,223,372,036,854,775,807$ ($-2^{63}+1$) to $9,223,372,036,854,775,806$ ($2^{63}-2$).

Percentage

Percentage is identical to Decimal number, except that the value is automatically multiplied by 100 and a % sign is added when displayed.

Date/Time

A column of data type Date/Time is internally stored as a Decimal number with the whole number portion representing the number of dates since December 30 1899 (which itself is stored as a value of 0) and the decimal portion representing

the parts of the day (with 0.5 representing 12 P.M., for example). The precision is 1/300 of a second.



Merging a column of type `Date/Time` that contains nonzero timestamps with a usual `Date` dimension won't succeed. The `Date` value is converted into a `Date/Time` with a time value of midnight (which won't match a nonzero timestamp of a column of data type `Date/Time`).

Date

`Date` stores only the date portion (not time portion). It's best practice to store only dates, or, if you need the time portion as well, store the time portion as a separate column.

Time

`Time` stores only the time portion (and no date portion).

Date/Time/Timezone

`Date/Time/Timezone` is `Date/Time` including the time zone. Power BI does not offer such a data type, so the column will be automatically converted into `Date/Time` when loaded into the data model.

Duration

`Duration` represents the length of time. Power BI does not offer such a data type, so the column will be automatically converted into `Decimal` number when loaded into the data model.

Text

`Text` is stored as Unicode (which allows for all sorts of special characters, including emojis) and can have a maximum length of 268,435,456 characters.

True/False

`True/False` can contain a Boolean value (or null).

Binary

A column of data type `Binary` can contain any data. It's best practice to either remove such columns or extract data out of them into one of the other available data types. A column of this type will *not* be loaded into the Power BI data model.

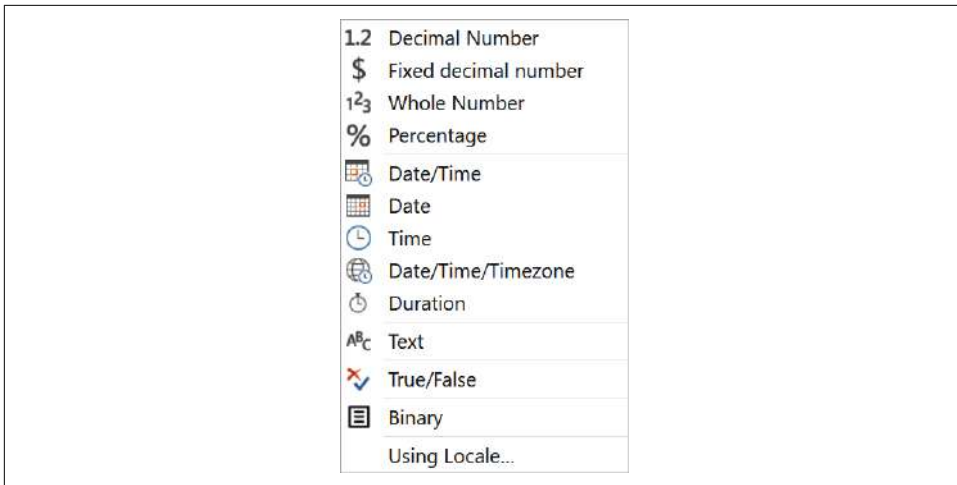


Figure 13-3. Data types in Power Query



If Power Query can't detect the data type and you don't set the data type explicitly, then a column will be of type Any. Such columns will be loaded as Text into the data model. It's best practice to be explicit about the data type of a column instead.

On top of these column types, the content of a column could also contain one of the following structures:

- A list of values
- A record, which is a grouping of a single row of data with individual fields
- A table contains multiple records
- An error

Typically, you see such columns when you load data from a JSON file or merge rows from a different query. All three types have in common that, to the right of the header of the column, Power Query shows an Expand icon with which you can extract individual values from the structures. If you decide against extracting the values, the content will not be loaded into Power BI's data model.

Next, I explain the role of relationships in Power Query.

Relationships

Because Power Query doesn't use the concept of tables, it doesn't employ the concept of defining relationships either. Still, a Power Query's result set ends up as a table in the data model. And therefore, one query can have a relationship to the content of a different query.

That's why Power Query lets you merge and append queries on the one hand and allows query references and adding and removing columns on the other hand (see [“Joins” on page 252](#) and [“Set Operators” on page 251](#)). The goal is to be able to combine individual queries into one or split a single query into several queries. If you split a query, you'll need to create a relationship later in the data model (to combine the information back again). If you combine queries, you will need fewer relationships.

Keep in mind that the star schema is the optimal way to split and combine queries. The number of relationships isn't so important, but the general design of the data model is. Use Power Query and M to shape the source data into a star schema. [Chapter 14](#) shows different techniques for how to achieve this in Power Query.

Primary Keys

If you load data into the data model with Power Query, it isn't important (or possible) that you identify the primary key column(s). Having a single column as the primary key is, however, important later in the data model. If you can't identify one column as the primary key, but only a combination of columns (composite key), then you need to concatenate those columns into a new column, which then forms the single-key primary key.

Concatenating the individual column values of a composite key into one column can be done via the UI: Ctrl-click the column headers in the desired order, then right-click and select Merge Columns (which will replace the existing columns; via the ribbon you can choose Merge Columns from either the Transform or the Add Column section, to either replace the existing columns with the merged result or add another column to the query). You can then choose a Separator and give the resulting column a “New column name” ([Figure 13-4](#)).

Figure 13-4. Merging several columns into one

Surrogate Keys

I very much prefer using surrogate keys to primary keys, which is also commonly considered best practice (see “[Surrogate Keys](#)” on page 8). But the question remains: does it pay off to add steps in Power Query to create surrogate keys (which slow data model refreshes) to replace nonnumerical keys from your data source? By choosing Add Column → “Index column,” you can easily introduce a surrogate key in your dimension tables. That’s simple. But you need the surrogate key in the (referencing) fact table as well. Therefore, you need to implement lookups (by choosing Merge and Expand in the ribbon, as I demonstrate in detail in “[Normalizing](#)” on page 44).

You can safely state that if a data model uses less memory, the report performance will be faster (as less memory has to be scanned to satisfy a query). That’s why I implemented two versions of the same data model for Power BI’s demo data, Financials:¹

- Dimensional model, based on the business keys (those without “surrogate key” in their name use the “business key”)
- Dimensional model, based on surrogate keys

With the help of DAX Studio, you have access to the VertiPaqAnalyzer functionality, which tells you how much space a table and/or column occupies in the data model.

¹ The examples in this section use the following files: *Financials Dimensional Model Surrogate Key.pbix*, *Financials Dimensional Model.pbix*, *Financials Filter Dimension Surrogate Key Measures.pbix*, *Financials Filter Dimension Surrogate Key.pbix*, *Financials Filter Dimension.pbix*, *Financials OBT Measures.pbix*, and *Financials OBT.pbix*.

Turns out that the difference in size for the Financials dataset is huge: creating the model based on nonnumerical business keys occupies roughly 10 MB of storage, while the same model, just based on surrogate keys, only occupies 277 KB. That's less than 3%! In other words, you can store 39 times as much information on any given infrastructure (your PC or Power BI Premium capacity) with the version of the model based on surrogate keys compared to the model based on business keys.

Taking a look at the details, the data itself is a very small size and almost the same in both approaches. The difference lies in the size of the *dictionary*, as shown in [Table 13-1](#). Using dictionaries is one of two ways of storing (and compressing) a column's value (also called *HASH* encoding, typical for texts; the other is *VALUE* encoding, typical for numbers).

The dictionary for the table `Discount Band` occupies 1 MB, while the data occupies only 264 bytes. This is, of course, an extreme case. The size of the dictionary is not dependent on the number of rows in the table, but dependent on the number of distinct values in a column (and has a minimum size of 1 MB). In a situation where you store not 700 rows, but 700 billion rows in the fact table for these four `Discount Band` values, the size of the dictionary will stay the same, and it'll probably be negligible compared to the size of the data.

Table 13-1. Model size comparison: business keys (BK) versus surrogate keys (SK)

Table	Cardinality	Size BK	Data BK	Dictionary BK	Size SK	Data SK	Dictionary SK	Size %
Financials	700	5,430,256	13,272	5,396,880	123,240	13,272	89,864	2.27%
Date	730	1,130,140	2,976	1,108,716	81,948	2,976	60,524	7.25%
Product	6	1,066,496	400	1,066,000	18,308	536	17,644	1.72%
Country	5	1,066,168	264	1,065,856	18,016	400	17,544	1.69%
Segment	5	1,066,168	264	1,065,856	17,996	400	17,524	1.69%
Discount Band	4	1,066,160	264	1,065,848	17,922	400	17,450	1.68%
TOTAL		10,825,388	17,440	10,769,156	277,430	17,984	220,550	2.56%

How much the refresh time increases due to introducing the index column in the dimension tables and looking up the results in the fact table is hard to predict as well. It depends on the number of rows in the dimension tables and the number of rows in the fact table compared to the available resources (memory and CPU). In the concrete example, in which the fact table has only 700 rows, the difference is negligible on my machine.

I return to this topic in [“Normalizing” on page 258](#), demonstrating how to further decrease the size of the data model by implementing a single filter dimension. Keep in mind that I suggest you consider replacing the business keys in the fact table with a surrogate key but keep the business key's content as a column in the dimension table if you need it in the reports.



Try out both approaches (keeping business keys versus introducing surrogate keys in Power Query) using your own data before deciding for or against one solution. The resulting data model's size (and the duration of the refresh) is pretty much impossible to predict because of the nature of the compression algorithm used in Power BI's VertiPaq engine, which is first and foremost optimized for fast queries not for predicting how much time compressing data will take or how much space the compressed data will occupy.

Surrogate keys or not—you will face situations where you need to combine several Power Query queries into a single one. Either you merge two queries in order to enrich the rows of the first one with additional columns, or you can just append one query to another. I talk about the latter in the next section.

Combining Queries

Queries (or better: their results) can be combined in two ways: with set operators (which basically add or remove rows) or with join operators (which basically add columns). Power Query allows you to show dependencies of such queries in a dedicated view.

Set Operators

The only set operator available in Power Query is UNION, which can be implemented via Home → Append Queries. In the dialog box, select if you want to append two queries to each other or more via radio buttons, and then select which queries you want to append from the dropbox (see [Figure 13-5](#)).

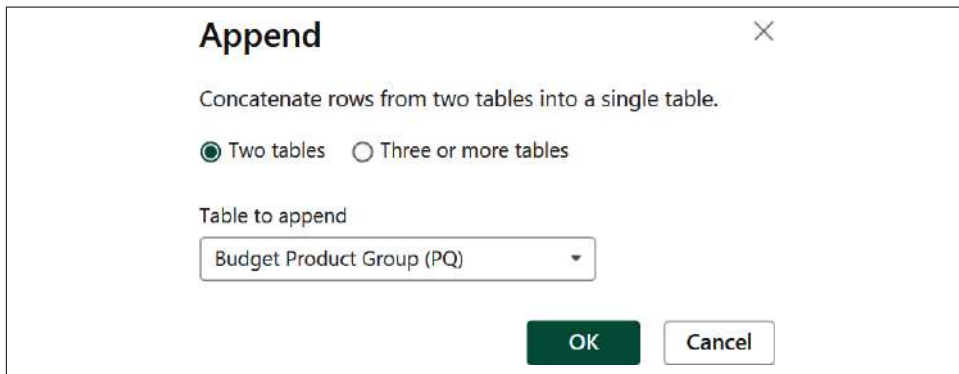


Figure 13-5. Append another query to the selected one

This step keeps duplicates, so you need to explicitly remove them if you don't want them (e.g., when creating dimension tables). You'll see examples of appended queries in [“Budget” on page 308](#), when you learn how to create a bridge table.

The next section is about joining tables, which is called *Merge* in Power Query.

Joins

You can join two Power Query queries via Home → Merge Queries. A dialog box will appear and display all the columns and the first four rows of the selected query (later, when choosing the “join” kind, this query is referred to as the *left* and *first* query). Then you choose the second table (referred to as the *right* and *second* query) via a dropdown. In [Figure 13-6](#), I show a merge query with DimProduct with DimProductSubcategory.

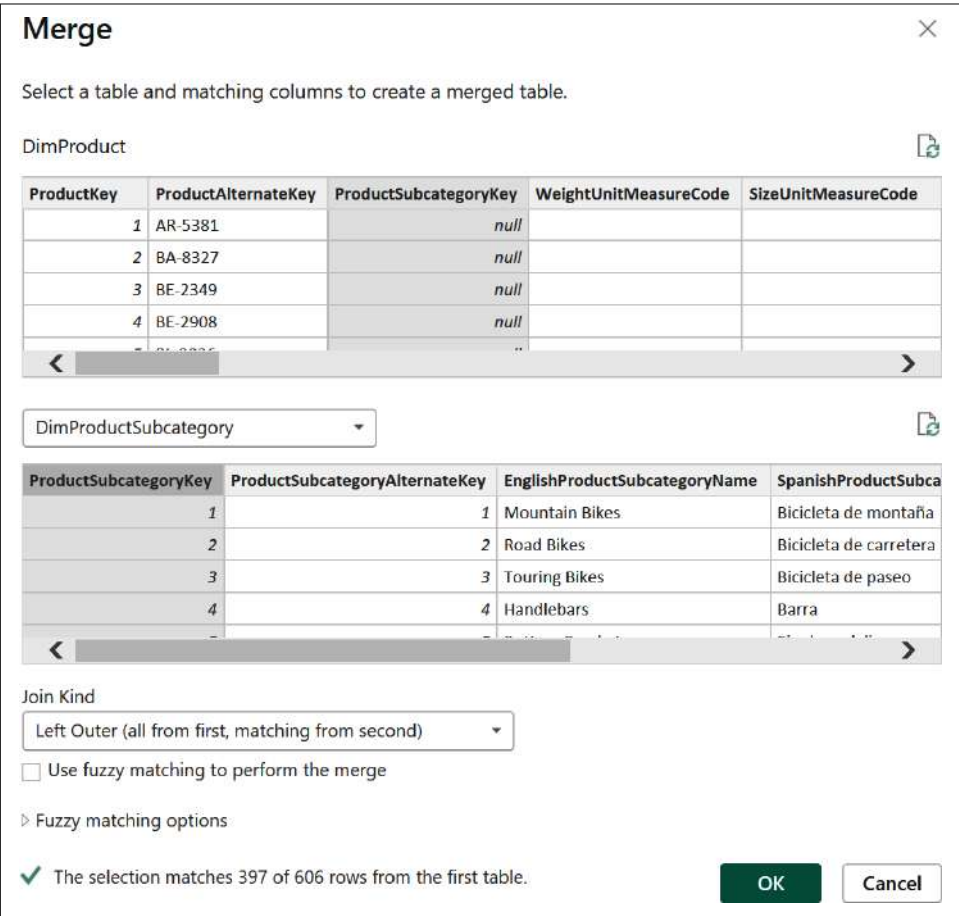


Figure 13-6. Merging two queries

Selecting the join predicate (the columns that should be used to join the two tables) is done by clicking on the columns. In case the join predicate consists of more than one single column, you can Ctrl-click the columns. Pay attention to selecting all the columns in the very same order for both the first and the second query, as this configures which columns are compared to each other. On the bottom of [Figure 13-6](#), you see how many matches the join predicate makes: *The selection matches 397 of 606 rows from the first table.*

The dropbox *Join Kind* ([Figure 13-7](#)) offers different kinds of joins, with a good description of which result to expect from each. All joins are equi-joins.

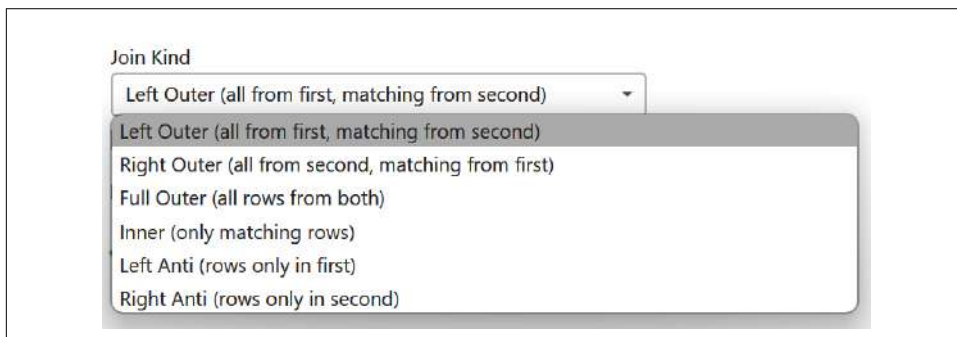


Figure 13-7. Different join kinds are available in Power Query

One join kind is missing in [Figure 13-7](#), however: a cross join. These joins are rare, but I demonstrate a use case in [Chapter 15](#). To implement a cross join in Power Query, you simply select “Add column” → Custom from the ribbon and just type in the name of the query you want to cross join with the current one.

The conditions of the equi-join can be loosened when you choose settings from “Use fuzzy matching” to perform the merge (see the bottom of [Figure 13-6](#)). Fuzzy matching allows you to fit two rows that are similar but not identical together on a column value. You need to find out for yourself if such an imprecise match will achieve what you are looking for, or if it would be better to clean up the data in the source system to allow for precise joins.

After you merge the queries, you need to expand the result and select the columns from the second query you want to add as columns to the first query. When you combine a lot of queries with each other, it is easy to lose perspective on which query is referenced with which. I talk about a way to keep an overview next.

Query Dependencies

In Power Query, you can visualize the dependencies between queries. Every query is dependent on at least one source. Queries can also be dependent on other queries. In complex scenarios Query Dependencies can give you a good overview. If you click

DimProduct, DimProduct and all the queries and sources on which it's directly or indirectly dependent on get highlighted (Figure 13-8):

- DimProductSubcategory is referenced directly in DimProduct. Therefore, a direct arrow is drawn between these two queries to visualize this direct dependency.
- DimProduct is indirectly dependent on DimProductSubcategory's source (i.e., `c:\users\mehe...`). This path isn't directly added somewhere in *DimProduct*'s query, but is part of the DimProductSubcategory, which is directly referenced in DimProduct.
- Direct dependency on its own source (`c:\users\mehe...`)

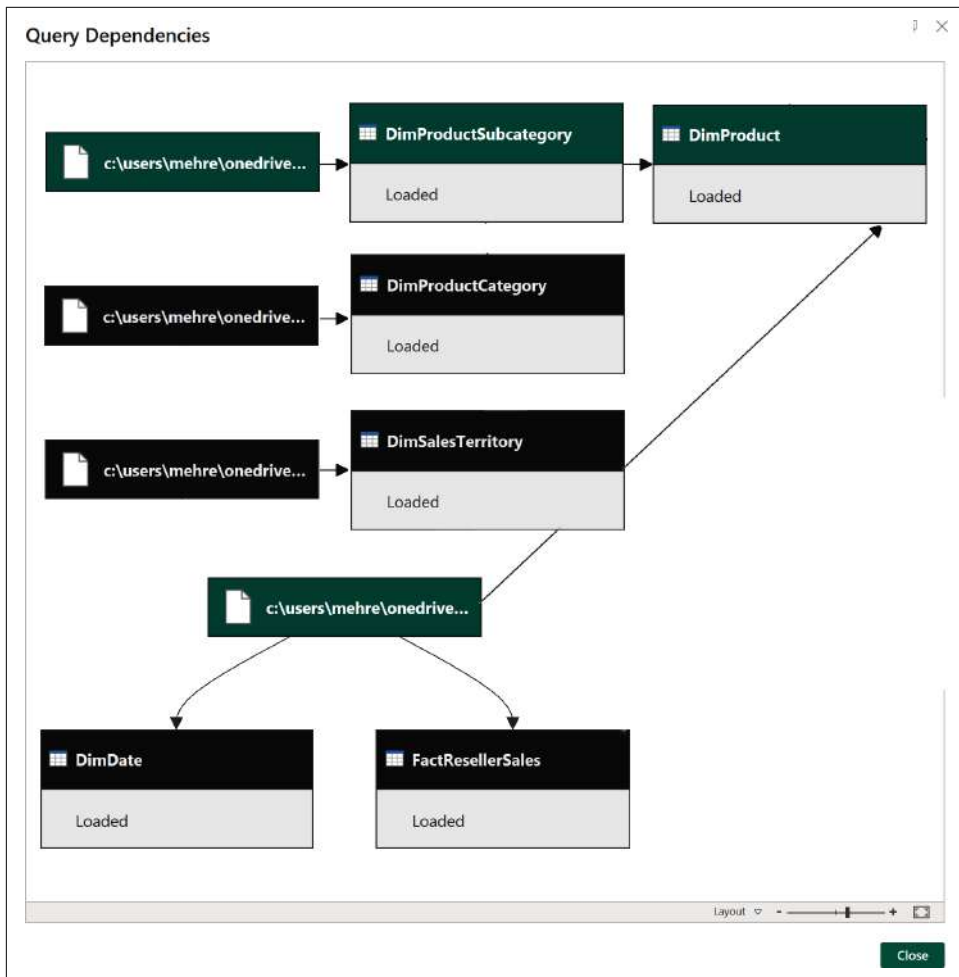


Figure 13-8. Displaying the dependencies between queries



Keep in mind that this is not about entities and their real-world relationships but queries and their technical relationships to each other. Therefore, this diagram is not an entity relationship diagram (ERD).

Types of Queries

Strictly speaking, Power Query has no tables (only queries), hence no type of table can be determined. But there can be different types of artifacts listed under queries:

Query

Every enabled query will load data into a table in Power BI's data model. It has at least one step (to reference the data source) and possibly a wide variety of additional steps to transform the data source into the desired shape.

Query (*"Enable load" disabled*)

Queries created only as an intermediate step (or for testing and debugging reasons only) should not be loaded into the data model. Therefore, you can right-click a query's name and disable the option "Enable load." The name of a disabled query is printed in italic font.

Parameter

Parameters come in handy, if you need certain values to be easy to access and change, or if you need a certain value be repeated in several queries or on several steps. Choose Home > Manage Parameters to manage the options of all parameters in a single dialog box, edit the value of the selected parameter, or create a brand new parameter. By default, a parameter's value is not loaded into Power BI as a table (if you choose to do so, it would appear as a table with one row and one column, containing the parameter's value).



Parameters of data type Any can't be changed in the Power BI service—avoid them and choose an appropriate data type instead.

Parameters are exposed in the Power BI service, which makes them a handy tool for specifying a data source's properties, like the server name or the filepath. The parameter's default value can be overridden in the Power BI service accordingly, to point it to the data sources in the production environment, for example. For parameters, the same data types apply as for columns.

Function

As in other programming languages, functions allow for code reuse. You specify the code only once (inside the function) and call the function whenever the code

should be executed. You can right-click any query and select *Create function* to create a function based on the selected query (the base query remains as it is).

Extract, Transform, Load

Power Query is the built-in self-service tool in Power BI to implement all steps of extracting, transforming, and loading data into the data model. It's easily accessible (as it's part of Power BI Desktop) and via the UI, you can clean and transform the data and achieve even complex steps without typing any code. For more advanced scenarios (e.g., because some features aren't exposed over the UI), you can also edit and write scripts in the Power Query/M language.

In [Chapter 15](#), we'll implement real-world use cases with the help of the UI and/or the scripting language to apply techniques to transform data in an efficient way.

Key Takeaways

Power Query is an integral part of Power BI—all the data in the data model must pass through Power Query. In general, Power Query is a better place for transformations compared to DAX for the following reasons:

- Power Query is the self-service tool inside Power BI to clean data and to bring it into the desired shape. I recommend using Power Query over DAX for those kinds of tasks, as it allows you to only load what is needed into the data model.
- Every Power Query query that uses “Enable load” becomes a table in the data model. The column's data types are the same in Power Query and Power BI.
- You can't create relationships in Power Query, but you can merge, append, and split queries so the resulting tables form the desired data model shape.
- Power Query is a low-code/no-code environment that enables complex transformations in the UI alone. Only in special cases will you need to write code in M, the Power Query mashup language. [Chapter 14](#) uses both the UI and M.
- Because you can neither read nor update the already existing rows of data during a refresh in Power BI, you can't implement slowly changing dimensions of any type in Power Query. If you need to implement slowly changing dimensions, you need a data warehouse. [Chapter 17](#) covers examples.

With the foundational knowledge about Power Query learned in this chapter, it's now time to apply it to build data model.

Building a Data Model with Power Query and M

Once you understand the moving parts of Power Query and the M language (see [Chapter 13](#)), it's time to see what this tool has in store for you to bring a data model into shape. You can achieve many tasks by finding your way around in the UI. No matter what kind of transformation you are applying, every step is “recorded” as a step in a script in language M. Think of this feature as something similar to Excel's macro recorder. Some steps will show a gear icon, which will open a dialog box and guide you through options to change a step's logic. You can also directly edit the scripts, if you want.

In this chapter, you'll learn how and why to touch some of the scripts directly. Editing the script is sometimes just faster, compared to navigating through the UI with the mouse. In other cases, it might be that a feature of the powerful language M is just not available via the UI. Especially when it comes to making the transformation either flexible or resilient against changes in the data source, M can do some magic.

You'll also learn how to normalize fact tables and denormalize dimension tables (to form a star schema) using Power Query. I'll introduce you to calculations in M and show you how you can transform flags and indicators into user-friendly text. A whole section is dedicated to creating a date table, which is mandatory for time-intelligence calculations in DAX and increases the usability of a data model by offering a variety of data- and/or time-related attributes (like year, month, weekday, etc.). The date table especially (but also other dimensions) may play different roles in your data model (e.g., to filter and aggregate by order date versus ship date). Creating these separate tables is rather easy in Power Query, as you will see. This chapter closes with a description of how to transform hierarchies such that they can be analyzed in Power BI. Let's start with normalizing.

Normalizing

You can normalize a table (and derive the necessary dimension tables) with just a few clicks in Power Query. But before you start, you need to find candidates for columns that should be normalized into one or more dimension tables. Power Query can assist you in finding these candidates, even before you load the data into Power BI (and create a report) as it can calculate descriptive statistics on the query's result.¹

Note the default setting: “Column profiling based on top 1000 rows.” It's a good choice performance-wise; it speeds up calculating the necessary statistics. Only if you doubt that the first thousand rows might not be representative of the full dataset should you change this setting to “Column profiling based on entire data set.” Then, Power Query will load the whole result set of the query into memory, which can take awhile, of course. You can change the setting by clicking the text on the bottom left of the Power Query window, as shown in [Figure 14-1](#).

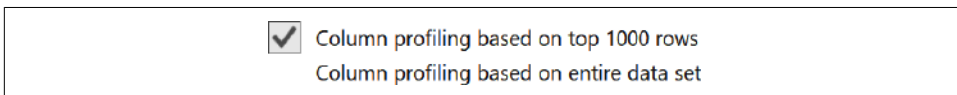


Figure 14-1. By default, column profiling is based on the top 1,000 rows

There are three statistics available:

- Column quality
- Column distribution
- Column profile

Column Quality

The setting View → “Column quality” activates the display of three indicators just underneath the column name, telling you which percentage of rows are Valid, have an Error, or are Empty. You may remember from “[Tables](#)” on [page 6](#) that primary keys must not be empty. It seems logical that a primary key must not evaluate to an error as well. This leaves the choice of a primary key with columns, which shows 100% for the *Valid* indicator.

If you're not confident that a column containing error or empty values is indeed the primary key, then you must first clean up the column to remove the errors and empty values. You could replace the erroneous or empty values or filter the rows out. The best option is to achieve this in the data source. The second-best option is to

¹ The examples in this section use the [Normalize Facts PQ.pbix](#) file in the book's GitHub repository.

transform the column in Power Query by replacing invalid values or filtering those rows out (if they're not needed). The column Segment shows 100% valid rows in [Figure 14-2](#).

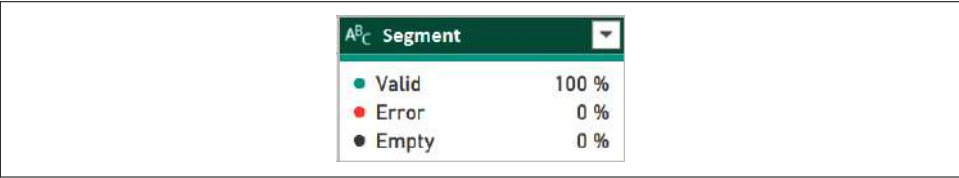


Figure 14-2. The quality of column Segment shows 100% valid rows

If you want to know what the valid values look like, you can turn on “Column distribution.”

Column Distribution

The option View → “Column distribution” activates a column chart (to be more precise: a *histogram*) just underneath the column name. This chart illustrates the distribution of the values in the column and two additional indicators: the numbers for *distinct* and *unique* values.

A column can only be a candidate to be a primary key if both *distinct* (number of different values in this column) and *unique* (number of values that only appear in one row for this column) are showing the total number of rows of the table. The column Segment in [Figure 14-3](#) shows five distinct values, none of which are unique. In the current table, Segment cannot be the primary key.

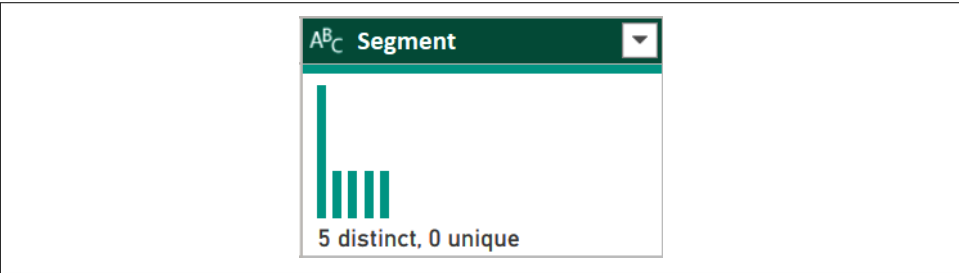


Figure 14-3. The distribution of column Segment shows five distinct and zero unique columns



The number of “unique” values in the column distribution shows how many of the values are only appearing once in this table. If “unique” is lower than “distinct,” some of the distinct values are duplicates. In other words, if “distinct” is lower than the number of rows, then not all rows can be “unique.”

For example, if a table has only three rows and one column and contains values 11, 12, and 13, then it has three distinct values (11, 12, and 13), and all three are unique (each value only appears once). If the column contains values 11, 11, and 12, then it would have two distinct values (11 and 12), and only one (12) would be unique.

A column containing the primary key must show the same number for “distinct” and “unique,” which should be the same as the number of rows in the table.

Column Profile

First enable the setting View → “Column profile,” and then click on a column. In [Figure 14-4](#), I selected the column Segment. The column profile displays the histogram from [Figure 14-3](#) as a bar chart and includes the distinct values of the column Segment. Additionally, it gives you descriptive statistics for the column’s content in the left section. The kind of statistics shown are dependent on the column’s data type.

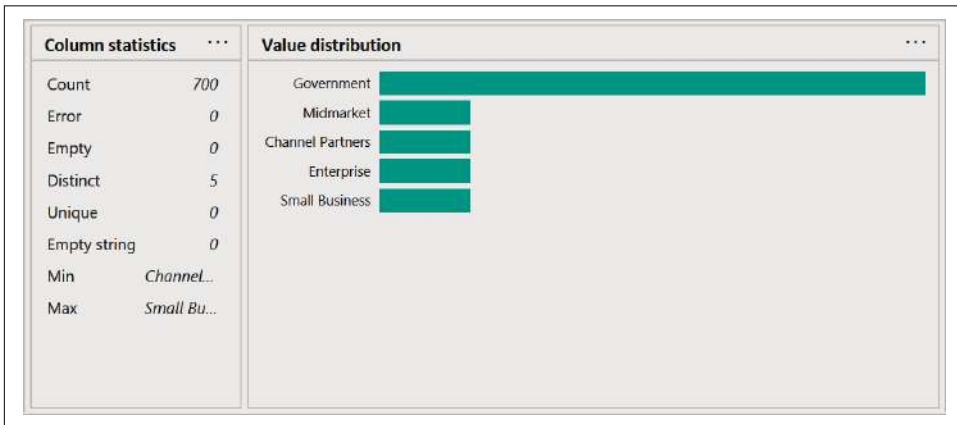


Figure 14-4. The column profile shows descriptive statistics and a histogram²

Via the ellipses (“...”) of “Column statistics,” you can copy all the information onto the clipboard. The same applies to the ellipses (“...”) of “Value distribution.” The latter allows you to “Group by” the histogram by different options. For example, you

² Print readers can find a full-sized, full-color version of this screenshot [online](#).

can group the Segment by text length instead of the actual values, as shown in Figure 14-5.

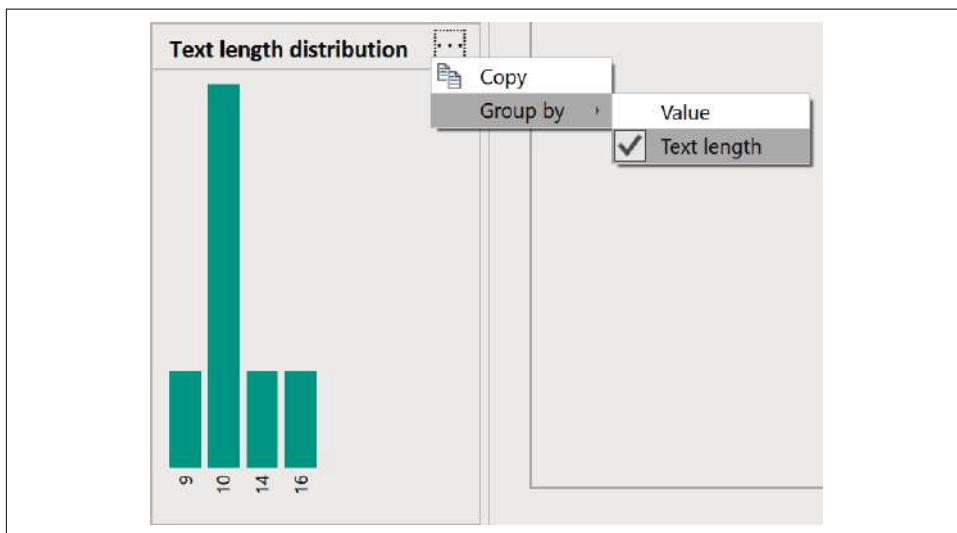


Figure 14-5. The histogram can be grouped by text length

The descriptive statistics of a column can give you a hint about whether the column should be normalized and will support you in identifying whether a column is transitive dependent on another column.

Identifying the Columns to Normalize

In the process of normalizing a fact table, you can use the statistical indicators column quality, distribution, and profile. Neither a primary key nor a foreign key may be empty or contain an error. The cardinality of such a column is low compared to the overall number of rows in a fact table.



In a star schema, a fact table is not (and shouldn't be) referenced by any other table. Therefore, you should omit the fact table's primary key if it's not needed in any report. Keep in mind that columns with high cardinality compress badly—keeping such a column is costly in terms of memory consumption. The rule here is “when in doubt, omit a column.” You can decide later to include the column, if really needed.

The columns you identify as foreign keys will become the primary keys for the dimension tables you “carve out” of the fact table during the process of normalizing. Foreign keys aren't required to be either distinct or unique, as they're on the “many”

side of the relationship. Foreign keys, though, typically have a low cardinality. For example, depending on your organization, your sales table might contain millions of rows, but only hundreds of products. The product is a dimensional attribute that needs to be stored separately from the fact table in a dimension table of its own. You need to walk through all columns whose content is not summable. Summable columns (like quantity or sales amount) stay in the fact table.

For numeric nonsummable columns, like price, you have two options: store the price in the fact table and/or move it into one of the dimension tables. If the actual price for a product can differ with every sale (e.g., because of discounts), then keeping them in the fact table is a good idea. If the list price for a product never changes, move it into the dimension table to save the space in the fact table.

To keep track of price changes over time in the dimension table, you need to implement slowly changing dimensions (see [“Slowly Changing Dimensions” on page 285](#)). If the goal is to store both the applied price and the list price, nothing will prevent you from storing the applied price in the fact table and still tracking the list price in the dimension table.

Let’s apply these ideas. For the following example, I use the built-in example file of Power BI Desktop (see [Figure 14-6](#)).

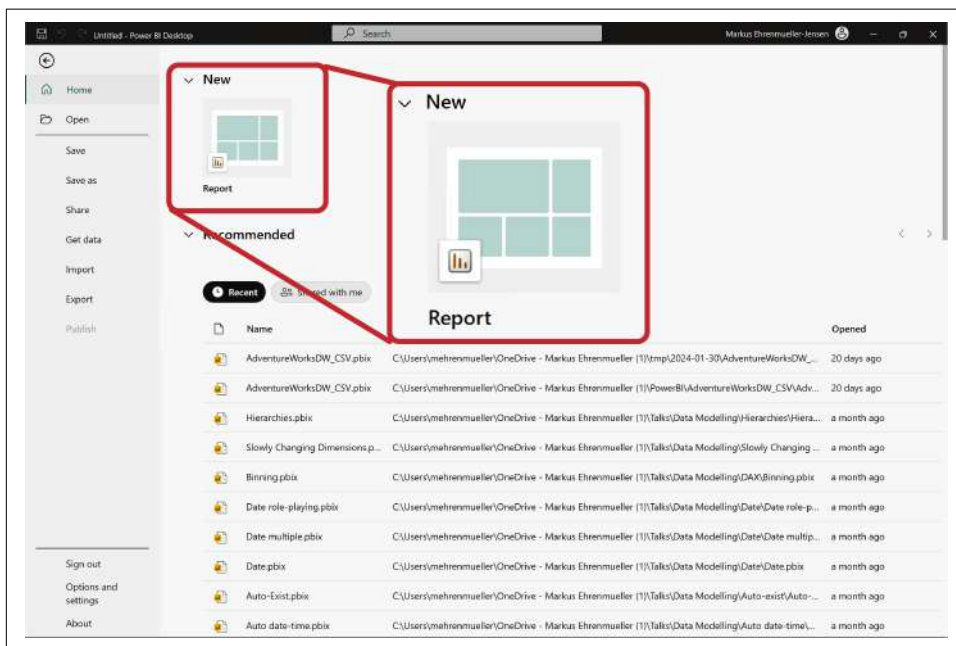


Figure 14-6. Power BI Desktop’s splash screen

Choose Report in the New section of the screen to get the choice of “Try a sample semantic model,” the right-most option on the canvas (Figure 14-7).

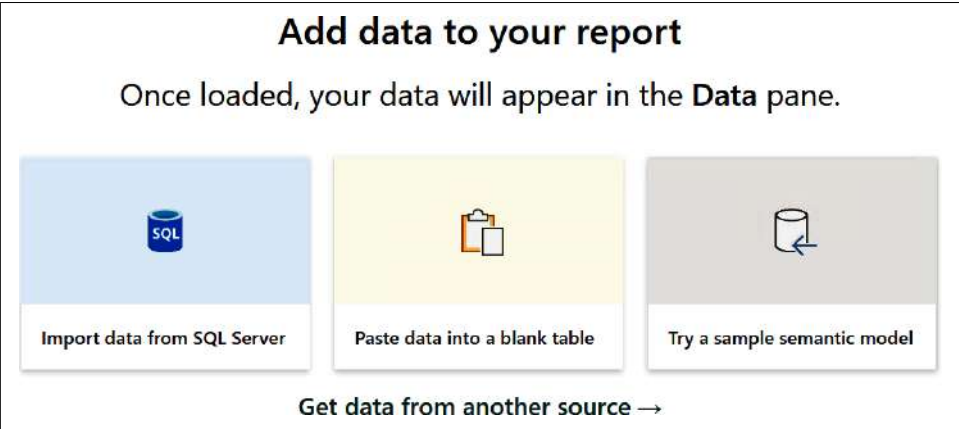


Figure 14-7. Try a sample semantic model

From the “Two ways to use sample data” dialog, choose the button “Load sample data” (Figure 14-8).

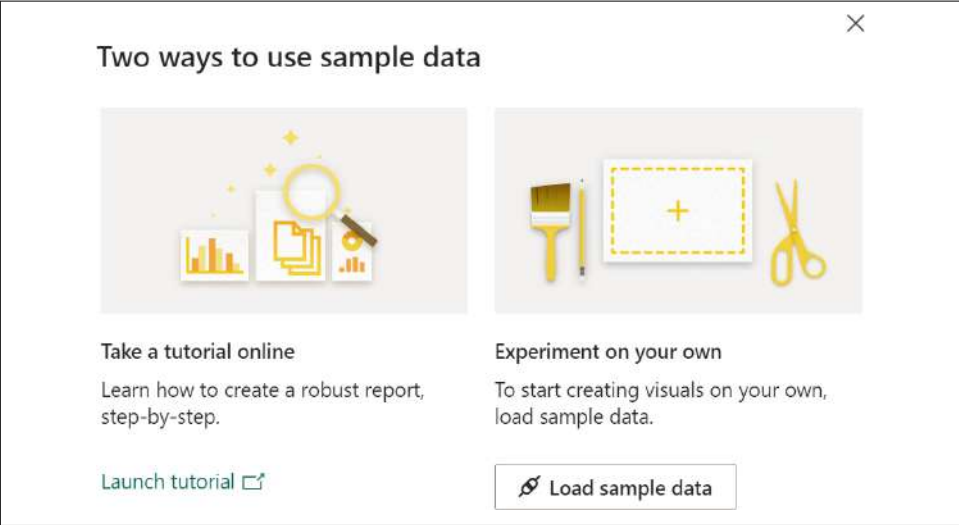


Figure 14-8. Two ways to use sample data

Then, a Navigator opens (Figure 14-9). If you’ve ever imported data from an Excel file, this screen will look familiar; it’s exactly the same. Please activate the checkbox to the left of `financials` and click Transform Data.

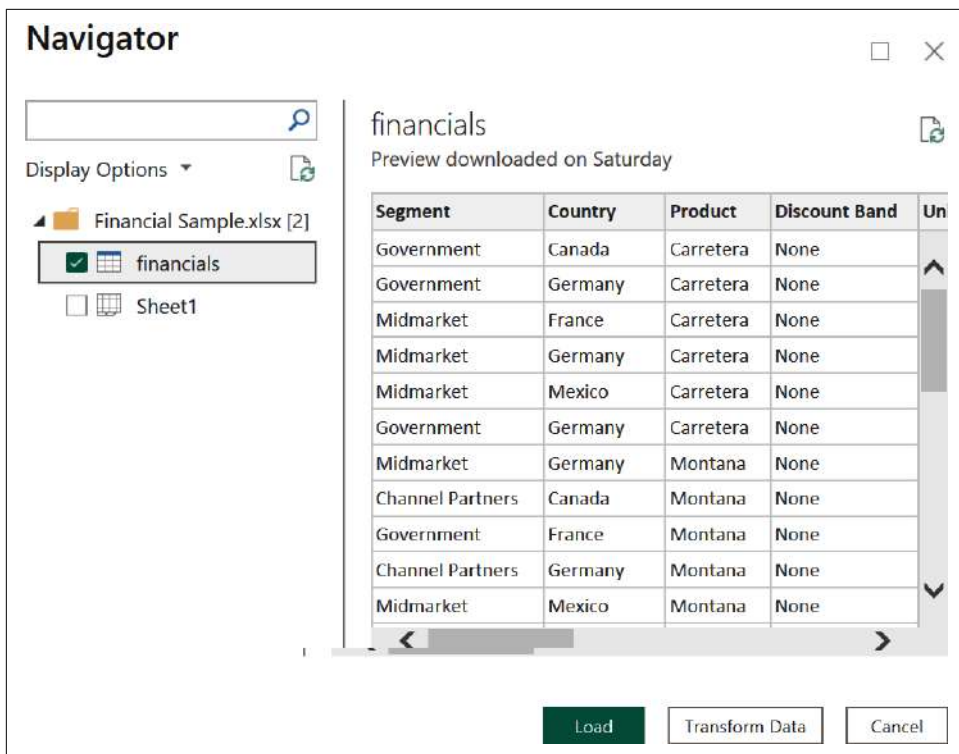


Figure 14-9. Choosing the needed artifacts from the Excel file in the Navigator



When you have the choice between Load and Transform Data, always choose “Transform data” (despite the fact that the Load button is highlighted). “Transform data” will open the Power Query window, and you can filter and transform the data, which is always necessary in a real-world scenario. I have never experienced a situation, where, e.g., an Excel file contains the data exactly as it is optimal for Power BI. Therefore, you should never expect to be able to just load it, but always plan to “Transform data” contained in the file.

The example file contains 700 rows—so the default of value of “top 1000 rows” will evaluate all rows for column profiling. In Power Query, you get the following results for the column distribution:

Column name	Number of distinct rows	Number of unique rows
Segment	5	0
Country	5	0
Product	6	0
Manufacturing Price	6	0
Discount Band	4	0
Sales Price	7	0
Units Sold	510	350
Gross Sales	550	406
Discounts	515	384
Sales	559	418
COGS	545	398
Profit	557	417
Date	16	0
Month Number	12	0
Month Name	12	0
Year	2	0

Some columns in this list are clearly candidates for dimensions, as their number of distinct rows is only a fraction of the total number of rows (700). `Sales Price` has a low cardinality as well, but I keep it in the fact table because it represents the price used for an individual sales event (as opposed to `Manufacturing Price`, which is tied to a product).

Creating a Query per Dimension

Before creating the queries for the dimensions, make sure that the `financials` query only contains cleaned data (e.g., naming is done properly, data types are correct, etc.). Then you have two options: you can either duplicate the query or reference it. Duplicate creates a full copy of the query, including all steps. In this case, I'd rather not duplicate those steps because if we discover a mistake later (e.g., incorrect column names or data types) I'd have to correct the mistake several times, once in each copy of the query. I strongly recommend referencing the query instead. This creates a new Power Query with the base query's name as the source step of the query ([Figure 14-10](#)). I give the referencing query then the name of the dimension table (e.g., `Segment`).

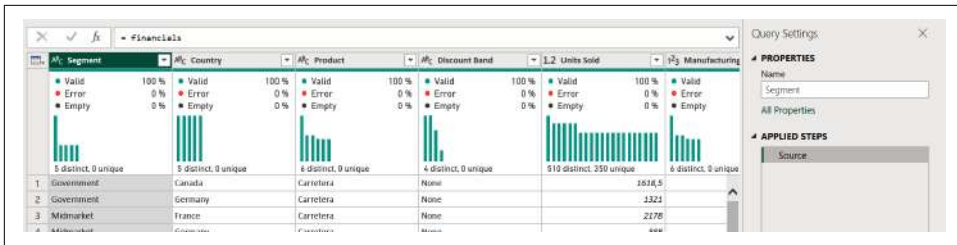


Figure 14-10. The Source step references query *financials*

Next, you need to remove all columns that shouldn't be part of the dimension:

1. Select Home → Choose Columns → Choose Columns from the ribbon or scroll through the query result in the middle of the screen (that's my preference because doing so shows me the values and the column profile).
2. Ctrl-click the columns you want to keep
3. Right-click one of the columns and select Remove Other Columns. If the table ends up containing more details for a dimension table later, you can always edit this step.
4. Choose either Advanced Editor from the View section of the ribbon, or turn on View → Formula Bar.
5. Edit the step and add or remove columns from the `Table.SelectColumns` function's parameter. Alternatively, you can click the gear icon next to the step Removed Other Columns in the APPLIED STEPS section and change your selection in a dialog box. See Figure 14-11.

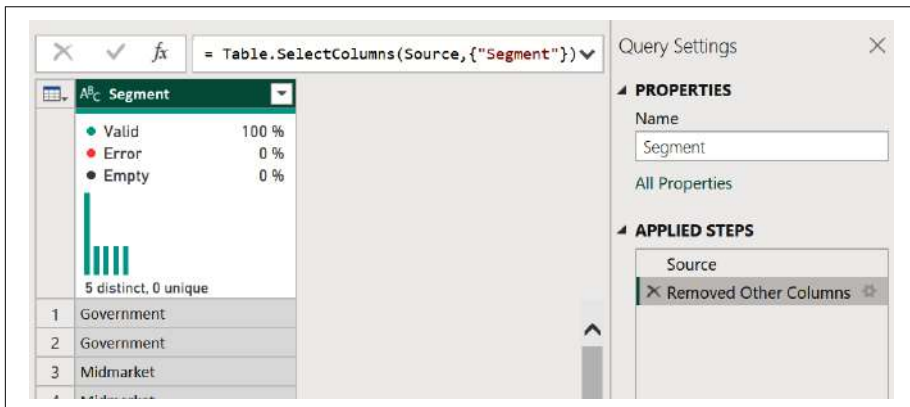


Figure 14-11. `Table.SelectColumns` function in the formula bar

6. Finally, you need to reduce the query to only distinct values, so that every unique dimension row only appears once in the dimension query. Just choose Remove Duplicates from the query's options (Figure 14-12).

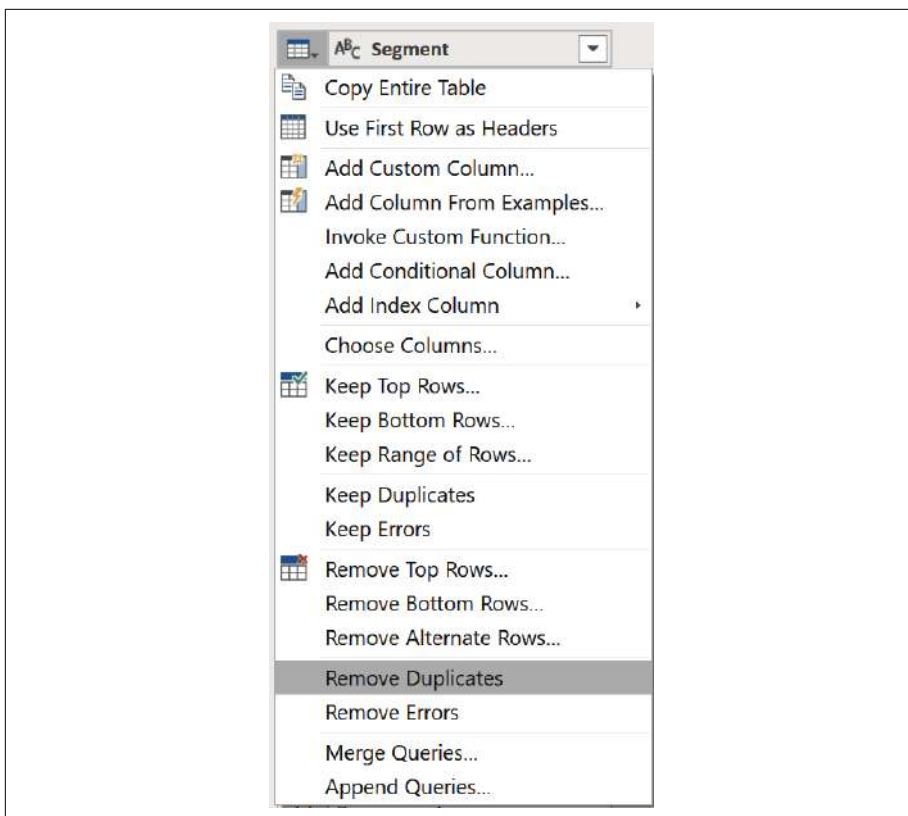


Figure 14-12. Removing duplicated rows from a query

Then, repeat all steps (referencing the base query in the Source step, selecting the dimensional columns and removing the duplicates) for all dimension candidates. The list of the applied steps will look the same for all those queries (see Figure 14-13).

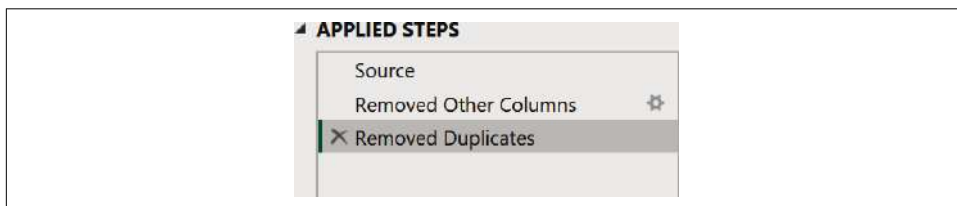


Figure 14-13. Creating dimensions in three steps

You can also directly copy and edit the M code generated from Power Query's GUI. Here is the script for the three steps:

```
let
    Source = financials,
    #"Removed Other Columns" = Table.SelectColumns(Source,{"Segment"}),
    #"Removed Duplicates" = Table.Distinct(#"Removed Other Columns")
in
    #"Removed Duplicates"
```

The Source step just references query financials. To remove the unnecessary columns, the column Segment is selected via Table.SelectColumns in the next step. Finally, Table.Distinct removes all duplicates, with only a few exceptions:

- One is the date table. You can't successfully derive the date table from the fact table; there aren't sales every single day of a year (e.g., the weekends, bank holidays, etc.), which is mandatory for a date table in Power BI. In ["Time and Date" on page 279](#), I show how you can create such a table in Power Query.
- In this example, it makes sense to store the manufacturing price only once per product. When you select the columns for query Product, make sure that you not only select the column Product, but the column Manufacturing Price as well. In general, for columns with the same statistical properties (distinct rows and unique rows), you should check if they can be part of the same dimension. Domain knowledge will guide you here. And you can create a temporarily Power Query query containing all those columns and see if you discover a functional dependency between these columns as soon as you ask to remove duplicates.
- The base query financials isn't appropriate as the fact table because it contains unnecessary columns (Manufacturing Price, Month Number, Month Name, and Year). Therefore, you need to reference the base query once more and remove these four columns. You only keep the references to the dimension's primary keys, and all factual attributes, of course. Make sure to *not* remove duplicates, as a fact table may contain duplicated rows, when the same product was sold twice on the same day to the same customer.

User-Friendly Naming Conventions

Because two Power Query queries (and tables in Power BI) can't have the same name, you need to either rename the base query (financials) or find a good name for the fact table. In my opinion, all tables (including the fact table) should have a user-friendly name that makes sense to people who use the data model to answer their questions. Therefore, I don't advise using prefixes like "Dim" or "Fact" in table's names (nor do I like camelCase, PascalCase, or hungarian_notation in such cases).

I use an underscore (_) as a prefix for names of objects (both tables and columns), which I hide from the users anyway. This means that I rename “financials” to “_financials” and I am free to keep “financials” as the fact table’s name (or any other name that makes sense to report creators).

Creating One Common Dimension Query

Alternatively, you can put all dimensional values together into one single filter dimension (see “[Normalizing and Denormalizing](#)” on page 109). To achieve this, take the following steps:

Creating a composite business key

In query _financials, you can concatenate the dimensions business keys into a new column (which will form a composite key). Just Ctrl-click the columns, and then right-click and choose “Merge columns.” Then choose a separator character for which you can guarantee that it will never be part of the individual columns’ content. Replace the default name Merged with _FilterKey (Figure 14-14). The M code looks like this:

```
= Table.AddColumn("#Changed Type", "_FilterKey",  
each Text.Combine({[Segment], [Country], [Product], [Discount Band]}, "|"),  
type text)
```

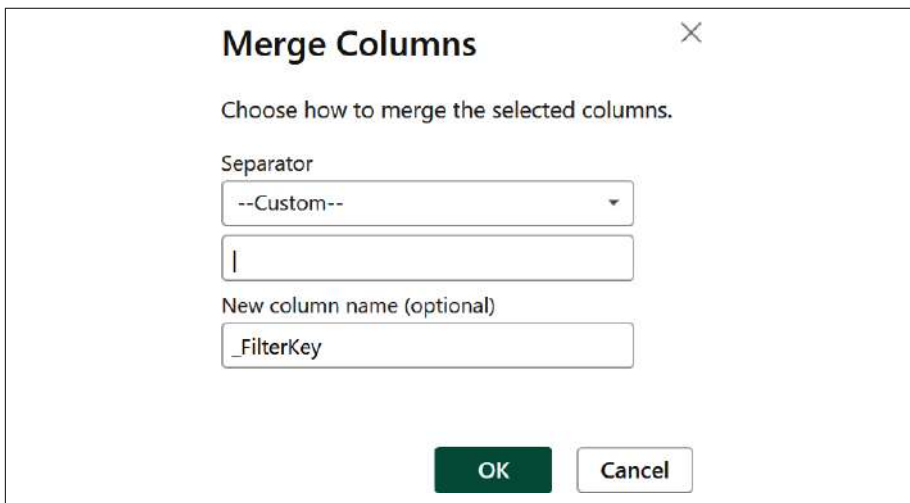


Figure 14-14. Merge Columns dialog, used to create a composite business key

Creating one dimensional table for all dimensional attributes

Here, keep all dimensional attributes instead of just one (as in “[Creating a Query per Dimension](#)” on page 265), including the business key column (_FilterKey).

Normalizing the fact table

In the fact table, you can remove all dimensional attributes and keep only the business key (`_FilterKey`).

The following hold regardless of the approach you choose: because you don't want to load both the original table and the newly created star schema, you should disable "Enable load" for the original table (`_financials` in the example). During normalization, remove columns containing duplicated information from the fact table.

In the next section, you'll learn how to denormalize tables in Power Query. This is the process of intentionally introducing duplicated information into a dimension table.

Denormalizing

Denormalizing means that you remove references from one dimension table and add the referenced attributes directly to the dimension table instead. In Power Query, you can achieve this by merging the referenced query with the referencing query. Select Home → Merge Queries → Merge Queries to add columns to the current query or Home → Merge Queries → "Merge Queries as New" to create an additional query (which I don't think is necessary in this case).



In Power Query, *merge* means that you add columns from one or more queries. You need to specify the merge key(s) to tell Power Query how it determines which rows shall be synchronized. You can specify a composite join key by holding the Ctrl key on the keyboard while selecting the columns with your mouse. Of course, the order of the keys matters—so you should click the columns in both tables in the same order. You can imagine merging as putting both queries side-by-side. In "[Joins](#)" on [page 13](#), I call this a *join*.

Append, on the other hand, puts the results of queries under each other. So, you add rows (and not columns). In "[Set Operators](#)" on [page 11](#), I refer to this as a *union* set operator. Columns with the same name in the source queries end up in the same column in the result. Columns available in one query, but not the other, will show empty values in the other queries.

In [Figure 14-15](#), I merge `DimProduct` with `DimProductSubcategory` on column `ProductSubcategoryKey` with a Join Kind of Left Outer to guarantee that all rows from `DimProduct` get loaded, even if there is no matching subcategory.³

³ The examples in this section use the *Denormalize Dimension PQ.pbix* file in the book's GitHub repository.

Merge

×

Select a table and matching columns to create a merged table.

DimProduct

ProductKey	ProductAlternateKey	ProductSubcategoryKey	WeightUnitMeasureCode	SizeUnitMeasureCode
1	AR-5381	null		
2	BA-8327	null		
3	BE-2349	null		
4	BE-2908	null		

DimProductSubcategory

ProductSubcategoryKey	ProductSubcategoryAlternateKey	EnglishProductSubcategoryName	SpanishProductSubcategoryName
1		Mountain Bikes	Bicicleta de montaña
2		Road Bikes	Bicicleta de carretera
3		Touring Bikes	Bicicleta de paseo
4		Handlebars	Barra

Join Kind

Left Outer (all from first, matching from second)

☐ Use fuzzy matching to perform the merge

> Fuzzy matching options

OK

Cancel

Figure 14-15. Merging ProductSubcategory into Product on ProductSubcategory Key

Implement Non-Breaking Changes in Views

If your data source is a table from a relational database that has foreign key constraints defined, you do not need to *Merge Queries* before you can expand the columns from the other table, but Power Query offers this directly to you.

I want to emphasize, though, that best practice is that you should not directly access tables but exclusively use views. Views give you an important extra layer. In case the structure of the tables changes, the views can be rewritten so that the schema of their result stays the same. This gives stability to your reports and relieves the pressure caused by changes to the tables and modifications in Power Query needing to be rolled out to production at the same time. Therefore, I see views as an API between the

database and Power BI. In a well-implemented data warehouse, non-breaking changes are done to the views (that means changes to existing columns like renaming, change of datatype, removing of columns). If breaking changes must occur, they have to be implemented as a new view together with a grace period to give the Power BI data modelers (you and me) time to implement the changes to the existing data models.

Then you expand the column via the icon just to the right of the column name (Figure 14-16).

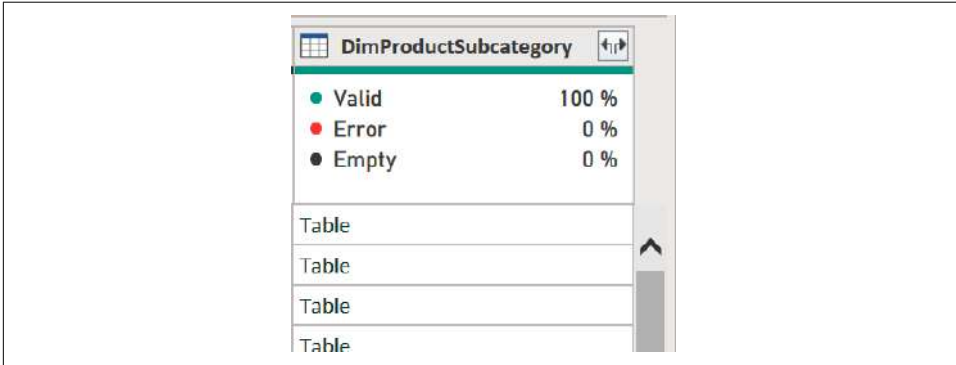


Figure 14-16. Column *DimProductSubcategory* can be expanded

Expanding allows you to choose which columns you want to expand and if you want to have a prefix in front of the original column names. In the example, I chose columns *EnglishProductSubcategoryName* (which is the name of the subcategory I want to show in my reports) and *ProductCategoryKey* (which I need to denormalize the category's name into the product table). As I want to keep the column names as they are, I chose to not have a “Default column name prefix” (Figure 14-17).

Don't forget to disable “Enable load” for the referenced queries, as all necessary columns are now part of a different query, and it does not make sense to load information twice into the data model. Abstain from actually deleting those queries, as this would break the merge step we just created.

The important steps in the M script are the following two:

```
#"Merged Queries" = Table.NestedJoin(#"Changed Type", {"ProductSubcategoryKey"},  
    DimProductSubcategory, {"ProductSubcategoryKey"}, "DimProductSubcategory",  
    JoinKind.LeftOuter),  
#"Expanded DimProductSubcategory" = Table.ExpandTableColumn(#"Merged Queries",  
    "DimProductSubcategory", {"EnglishProductSubcategoryName",  
    "ProductCategoryKey"}, {"EnglishProductSubcategoryName",  
    "ProductCategoryKey"}),
```

Function `Table.NestedJoin` joins the two queries in a left outer fashion. `Table.ExpandTableColumn` extracts the two columns (and keeps their original name).

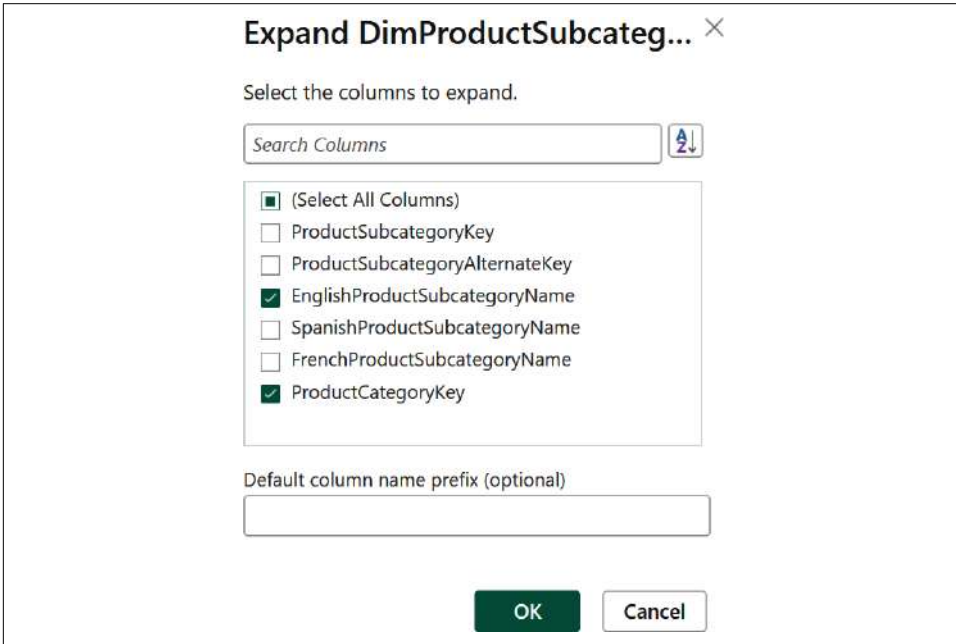


Figure 14-17. A choice of columns to expand

Not many people are eager to build reports that show long lists of data. Every report I've built so far contains at least one measure that, in its simplest form, aggregates individual values. In the next section, I introduce you to calculations in Power Query.

Calculations

In Power Query, you can implement numerical calculations of high complexity. Remember, though, that it only makes sense to add calculations for which the result is fully additive in a report. A visual rarely shows individual rows of a table, but aggregated values in most cases. For example, the count of rows is an additive calculation; the distinct count definitely not. While I, in general, do not recommend adding transformations in DAX, I fully recommend adding calculations (in the form of DAX measures). As even additive calculations can be done in a DAX measure, I wouldn't create calculations in Power Query but start first in DAX. Only if you see performance problems should you move the formula to Power Query and thereby persist (intermediate) results. The size of your data model will grow when you add new columns, and the speed of the report creation might increase.

You can use a calculation to replace the value of an existing column (Transform) or by adding a new column (“Add columns”). Both the Transform and the “Add columns” sections in the ribbon of Power Query offer a wide range of transformations, including calculations on numeric values (Figure 14-18):

Statistics

Sum, Minimum, Maximum, Median, Average, Standard Deviation, Count Values, Count Distinct Values

Standard

Add, Multiply, Subtract, Divide, Integer Divide, Modulo, Percentage, Percentage of

Scientific

Absolute Value, Power, Square Root, Exponent, Logarithm, Factorial

Trigonometry

Sine, Cosine, Tangent, Arcsine, Arccosine, Arctangent

Rounding

Round up, Round down, Round specific decimal places

Information

Is Even, Is Odd, Sign

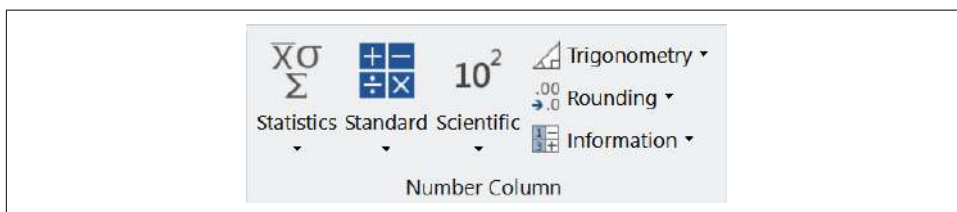


Figure 14-18. The Number Column section in Power Query’s ribbon

Calculations in Power Query are transformations of numeric columns. You can apply transformations to nonnumerical columns as well, as I show next.

Flags and Indicators

Transforming and adding columns is not only possible with numeric columns (as the previous section describes) but is an easy task for any type of column with Power Query. In the following examples, you will learn about the strength of Power Query when it comes to transformations of any kind. I will show you which button to click in the UI, but we will talk about the generated M code as well.



Power Query's UI is very user-friendly, and you can achieve many powerful transformations without writing a single line of code. You can go far in Power Query without touching (or understanding) M. Only on special occasions do I turn toward the M code: fixing a typo in text I provided, duplicating logic (e.g., the rules of a conditional column) by copying and pasting existing parts of the code, or if the UI doesn't provide a functionality (yet).

Chapter 15 cover advanced solutions with examples. Here, I want to open your mind to M by demonstrating both the UI and the M code behind.

In the following list, I apply a (slightly) different method of transformation on each column:⁴

FinishedGoods

To transform column `FinishedGoods` (which contains 0s or 1s) to a descriptive text column, I select **Add Column** → **Conditional Column** from the ribbon. Then I type in **Finished Goods** as the “New column name” and provide the rules to transform a value of 0 to “not salable” and a value of 1 to “salable.” In the Else field, I provide “unknown,” as you can see in **Figure 14-19**.

If you make sure that **View** → **Formula Bar** is enabled, then you can read the generated M code when you click **Added Finished Goods** under “Applied steps”:

```
= Table.AddColumn(Source, "Finished Goods", each
    if [FinishedGoodsFlag] = 0 then "not salable"
    else if [FinishedGoodsFlag] = 1 then "salable"
    else "unknown"
)
```

⁴ The examples in this section use the *Flag.pbix* file in the book's GitHub repository.

Add Conditional Column

Add a conditional column that is computed from the other columns or values.

New column name

	Column Name	Operator	Value ①		Output ①	
If	FinishedGoodsFlag	equals	ABC 123 0	Then	ABC 123 not salable	...
Else If	FinishedGoodsFlag	equals	ABC 123 1	Then	ABC 123 salable	

Add Clause

Else ①
ABC 123 unknown

OK Cancel

Figure 14-19. You can provide rules to create a new column based on the values of an existing column

Style

In general, I try to avoid solutions with conditional columns/nested if statements because I (or some poor other person) would need to dig into the code to find where the condition is hidden if the logic has to be changed (e.g., an additional value becomes available or the descriptive text has to be adapted). I prefer to have a lookup table instead. In this example, I create table `MyStyles` via Home → Enter Data and provided a column named `Style` with the style's code (W, M, U, and an empty value) and a column `StyleDescription` with the descriptive text, as shown in [Figure 14-20](#).

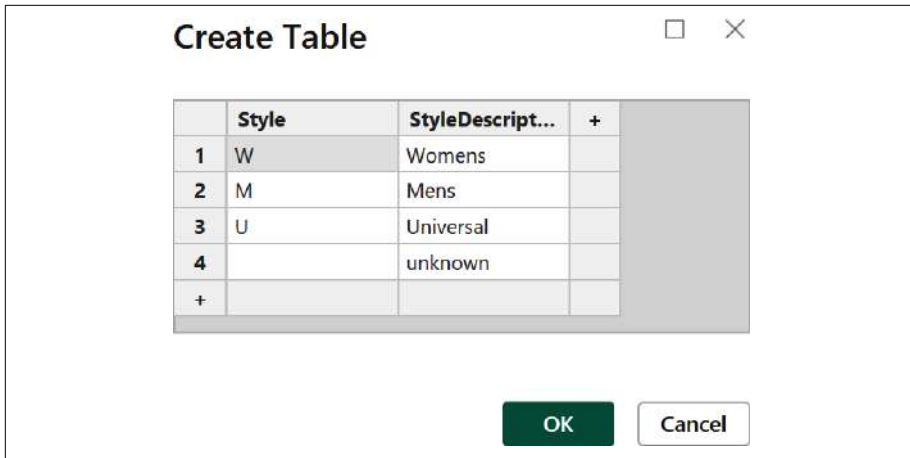


Figure 14-20. A table containing each style and its descriptive text

Editing this table is usually faster and easier than changing code containing if statements or filling out the Add Conditional Column form, offered in a dialog via the gear icon of the Added Finished Goods step. To change the content of the table, just click the gear icon to the right of the step in “Applied steps.” In a perfect world, the MyStyles table would be provided from outside Power BI, where the responsible users (with the business’s authority) have editing rights. This could be anything: an Excel file on a shared OneDrive, a SharePoint List, or a table in a database, which the users can edit via an application.

After you create the lookup table, you need to merge it into the existing query (Home → Merge Queries → Merge Queries) and expand the newly added column to add a StyleDescription (Figure 14-21).

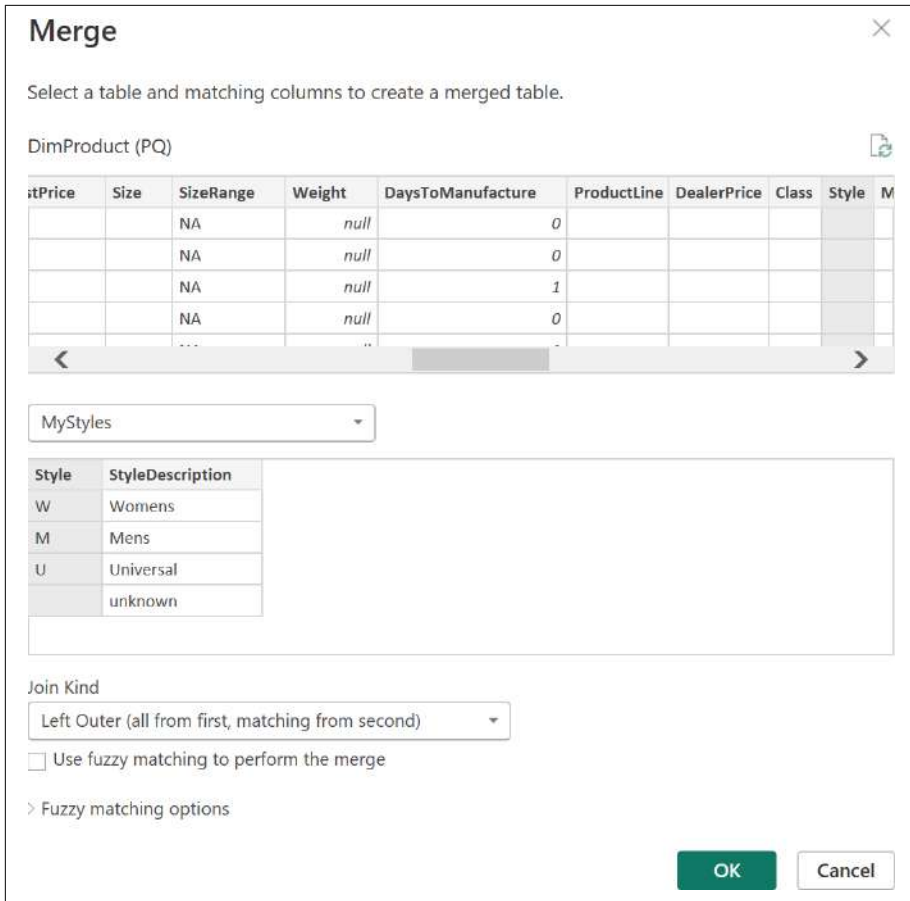


Figure 14-21. Merging the created table into DimProduct

Because column `Style` in table `DimProduct` contains blanks, and Power Query will respect those blanks while merging tables, I add an extra step, `Trimmed Style` just before step `Merged MyStyles`: right-click the column's header and choose `Transform` → `Trim`. And here's the M code that was generated via the GUI:

```
#"Trimmed Style" = Table.TransformColumns(#"Added Finished Goods",
    {{"Style", Text.Trim, type text}}),
#"Merged MyStyles" = Table.NestedJoin(#"Trimmed Style", {"Style"},
    MyStyles, {"Style"}, "Styles", JoinKind.LeftOuter),
#"Expanded Styles" = Table.ExpandTableColumn(#"Merged MyStyles", "Styles",
    {"StyleDescription"}, {"StyleDescription"}),
```


WeightUnitMeasureCode

Replacing a value in a column can be done by right-clicking on a column's value and choosing Replace Values. Value To Find will already contain the selected value. You just fill out Replace With accordingly. That's how I replaced an empty WeightUnitMeasureCode with N/A. Column WeightUnitMeasureCode in table Dim Product is sometimes empty (which is a blank string in the UI and two double-quotes, "", in M), as shown in [Figure 14-22](#). In other situations, a value, independent of the datatype, could also be null. And "null" is exactly what you would type into the field in the UI and/or in M code if you want to replace null or replace something with null.

```
= Table.ReplaceValue("#Expanded Styles", "", "N/A", Replacer.ReplaceValue, {"WeightUnitMeasureCode"})
```

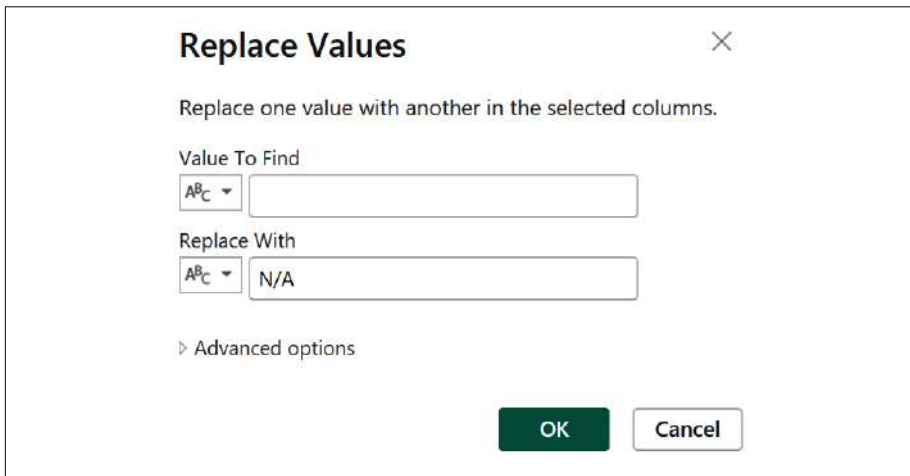


Figure 14-22. Replacing empty values with N/A

Time and Date

In “[Normalizing](#)” on page 258, I promised to show you how to create a date table in Power Query. Now the time has come!

To generate a time and a date table from scratch in Power Query, you need to reach out to the M language; you can't completely rely on the UI. In the first step, you need to generate a list of dates (or timestamps), which the dimension table should cover.

Start with Home → New Query → Blank Query ([Figure 14-23](#)).⁵

⁵ Use *Date.pbix* to follow along with the examples in this section.

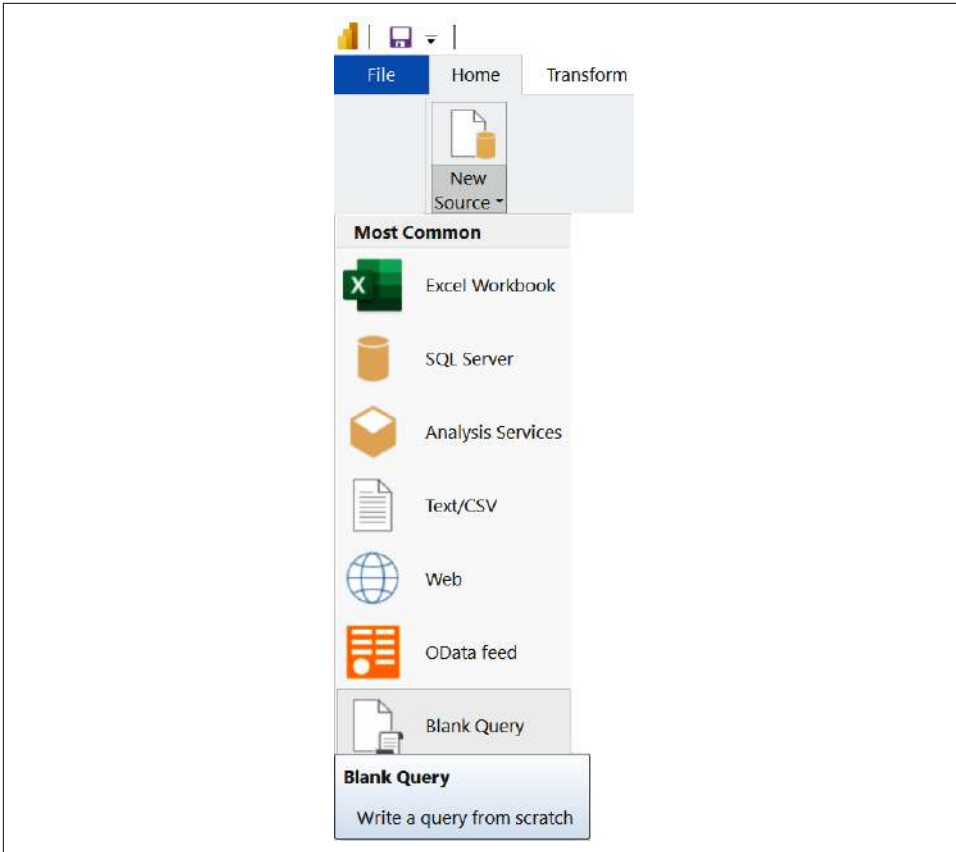


Figure 14-23. Starting with a Blank Query in order to provide M code

In the following code example, I start with Power Query's *list indexer operator* (`{}`). Inside the curly braces, I pass in the start value, then two dots (`..`), and then the end value. In this case, the operator only accepts numeric indexes but not a date or timestamp. Therefore, I use function `Number.From` to convert the two dates into a number (representing days since December 30 1899). I resist providing a hard-coded time and date but used lookup column `OrderDate` from query `Fact Reseller Sales`. This gives me peace of mind if data for a new year is available or old data is removed. As I refer to the fact table's content, I can be sure that the date table will always cover the full time range. For the start date, I apply `Date.StartOfYear` and `Table.Min`. For the end date, `Date.EndOfYear` and `Table.Max`, respectively:

```
= {Number.From(Date.StartOfYear(Table.Min("#Fact Reseller Sales", "OrderDate")
[OrderDate])) .. Number.From(Date.EndOfYear(DateTime.Date(Table.Max(
"#Fact Reseller Sales", "OrderDate")[OrderDate])))}
```

If you then right-click the header (List), you can convert To Table, as shown in Figure 14-24.

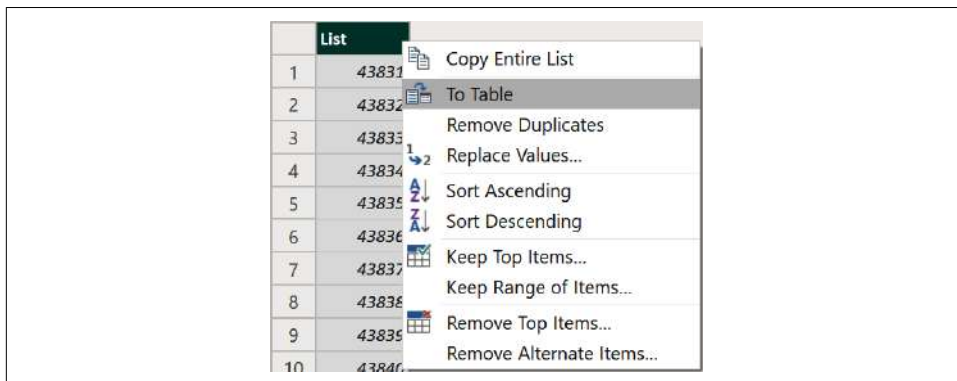


Figure 14-24. Converting a list to a table

Here I insert the M code, which was generated by the To Table step:

```
= Table.FromList(Source, Splitter.SplitByNothing(), null, null,
  ExtraValues.Error)
```

To the left of Column1, you can click the button to select the correct data type, Date for the column (Figure 14-25).

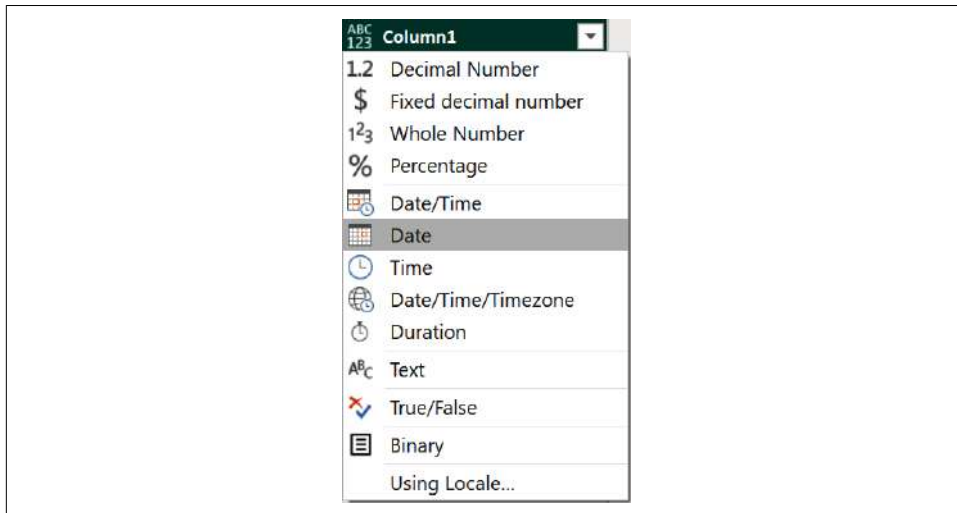


Figure 14-25. Setting the right data type for the column

Table.TransformColumnTypes is the corresponding function in M:

```
= Table.TransformColumnTypes("#Converted to Table",{{"Column1", type date}})
```

Column1 isn't so practical as the column's name. Double-click the header and type in **Date**. The resulting code looks like this:

```
= Table.RenameColumns("#Changed Type",{{"Column1", "Date"}})
```

Then I add informative columns to this query. Select the Date column and choose from the available transformations under Add Column. I choose the following:

Date → Year → Year (e.g., 2023):

```
= Table.AddColumn("#Renamed Columns", "Year", each Date.Year([Date]), Int64.Type)
```

Date → Month → Month (e.g., 01):

```
= Table.AddColumn("#Inserted Year", "Month", each Date.Month([Date]), Int64.Type)
```

Date → Month → Name of Month (e.g., January):

```
= Table.AddColumn("#Inserted Month", "Month Name", each Date.MonthName([Date]),  
type text)
```

Date → Day → Day (e.g., 31):

```
= Table.AddColumn("#Inserted Month Name", "Day", each Date.Day([Date]),  
Int64.Type)
```

Add as many versions and combinations as will be useful for your report users. This table will contain a maximum of 366 rows per year. If your report covers 10 years, this table will contain fewer than 4,000 rows. Therefore, you can easily afford to enrich it with columns to support every potential need of the report creators.

A numeric key for the table can be useful in some situations. For the date table, I prefer a numeric key, which represents the date in a YYYYMMDD fashion (20230801 would be the key for August 1, 2023). I add the following step to my query:

```
= Table.AddColumn("#Inserted Day", "DateKey", each Text.Combine({  
Date.ToText([Date], "yyyy"), Date.ToText([Date], "MM"), Date.ToText([Date],  
"dd"})),  
type text)  
= Table.TransformColumnTypes("#Added Custom Column",{{"DateKey", Int64.Type}})
```

The solution for generating the Time dimension from scratch in Power Query involves similar steps. You start again with Home → New Source → Blank Query and use List.Times to generate a list of timestamps.

As the first parameter, I pass in #time(0, 0, 0), which represents a timestamp for midnight (0 hours, 0 minutes, and 0 seconds). As the second parameter, I pass in the expression 24 * 60, which gives the number of rows I want the query to have (24 hours per day and 60 minutes per hour to cover a time table for every minute of the day). You could just write 1,440 (the result of the expression), but I find it more readable to provide the calculation. The third parameter provides the steps the final result

should contain. I use `#duration(0, 0, 1, 0)` because I want a table for every minute (0 days, 0 hours, 1 minute, 0 seconds).

Here, you see the line of code for the Source step of the query:

```
= List.Times(#time(0, 0, 0), 24 * 60, #duration(0, 0, 1, 0)) -- 24h * 60minutes
```

The next steps include the conversion of the list into a table, changing the data type of the column to `Time` and renaming the first column to `Time`. I won't repeat the description of the UI here; it's very similar to what we did when creating the date table. Here is the M code:

```
= Table.FromList(Source, Splitter.SplitByNothing(), null, null,
  ExtraValues.Error)
= Table.TransformColumnTypes(#"Converted to Table",{{"Column1", type time}})
= Table.RenameColumns(#"Changed Type",{{"Column1", "Time"}})
```

You can add variations of the timestamp via Power Query's UI. Choose **Add Column** in the ribbon and then:

Time → Hour → Hour (e.g., 11):

```
= Table.AddColumn(#"Renamed Columns", "Hour", each Time.Hour([Time]), Int64.Type)
```

Time → Minute(e.g., 59):

```
= Table.AddColumn(#"Inserted Hour", "Minute", each Time.Minute([Time]),
  Int64.Type)
```

I also want a column containing the time as a string in the format `HH:MM`. Let's look at Power Query's feature **Column From Examples**.

First, Ctrl-click both the `Hour` and `Minute` columns. Then choose **Add Column** → **Column From Examples** → **From Selection**. Now you need to "teach" Power Query by giving examples of the results you want to achieve by a Power Query expression. Double-click in column `Column1` on the line for `Time` `"00:01:00"` and type in `"00:01."` Power Query will fill out the rest of the rows.

By the unintended leading zeros in the results, you can see that Power Query didn't fully understand this. You'll have to provide another example to correct it. Double-click `"00:010"` and change it to `"00:10"`. There is another incorrect value, `"10:00"`, which we correct to `"01:00"`. When using this feature, always make sure to look out for weird results (and correct them). Make sure that you understand the code at the top of the screen. Here are my results after making corrections:

```
= Table.AddColumn(#"Added Custom Column", "Custom", each Text.Combine({
  Text.PadStart(Text.From([Hour], "en-AT"), 2, "0"), ":",
  Text.PadStart(Text.From([Minute], "en-AT"), 2, "0"))}, type text)
```

The formula concatenates the hour with a colon (:) and the minute. For both the hour and the minute, it makes sure to add a leading "0" and then chooses the two right-most characters. This looks perfect to me. You can keep the formula. Before

you press enter, double-click the header and provide `Time Description` as the column's name.

It is not unusual for a dimension to play several roles inside a data model. And this is also very typical for the date table. The next section describes how to do this in Power Query.

Role-Playing Dimensions

A dimension in a data model can play different roles in different contexts; you may have both an order date and a sales date in your fact table. As [“Role-Playing Dimensions” on page 123](#) points out, you can either create several filter relationships between the fact table and the date dimension, or you can add the date dimension several times into the data model under different names (e.g., `Order Date` and `Sales Date`).

You can easily achieve the latter in Power Query. Simply right-click the query that has multiple roles in the data model and choose either `Duplicate` or `Reference`. If you duplicate a query, the new one will contain duplications of all the steps of the original query. Referencing means that the new query only contains one step, which references the other query. This might not seem like much of a difference, but if you need to change some of the steps later, the difference will become clear. If the changes should be applied to all queries (the original one and the duplicates), you need to repeat the change for all queries. Referenced queries will automatically receive the changes.

Deciding between `Duplicate` and `Reference` is a bet on the future: how high is the probability that future changes will need to be applied to all copies of the query? If the probability is high, then `Reference` is the better choice. How high is the probability that changes will only need to be applied to individual copies of the query? If the probability is high, then you're better off duplicating all steps, generating an independent query.

When it comes to role-playing dimensions, I usually decide to apply all necessary steps (transformations like renaming of columns, setting the correct data type, etc.) to the original query and then reference the query as many times as I have roles for this dimension. (This is similar to the workflow for creating dimension tables out of a fact table in [“Normalizing” on page 258](#).) Make sure to rename the queries accordingly (e.g., copies of the `Date` query become `Order Date` and `Ship Date`).

Pay attention to the fact that the two queries are now fully identical—except for their names. When you load these queries as they are, you end up with columns with identical names appearing in multiple tables. That's very confusing if you search for something in the data model, which undermines our ultimate goal of making the report creator's life easy. It's also confusing for the report consumer because the

column names become the standard header for the visuals. If the report creator doesn't take the time to change those default headers, it's up to the report consumer to guess if "Year" means the year from the Order Date or the Ship Date table. Before you start manually renaming columns, take another deep breath and review the following M code:

```
= Table.TransformColumnNames(Source, (columnName as text) as text => "Order " & columnName)
```

This small line of code iterates over all columns (not the rows!) of the existing query and adds the text Order in front of the column name. The code is fully independent of the number of columns or the actual name of the existing columns—and makes it, therefore, resilient to changes in the table. I added the same code to the Ship Date (PQ) as well (with Ship instead of Order, of course). This shows the beauty of M code: you can make transformations dynamic instead of statically renaming every single column.

Dimensions don't just play different roles; their attributes can change as well. If you want to track those changes you need to learn about the concept of slowly changing dimensions.

Slowly Changing Dimensions

The idea behind the different types of slowly changing dimensions is to track changes. Depending on the desired type, you need then to update existing rows or insert additional rows in the existing tables. None of that is possible with Power Query.

With the way Power BI (and the storage engine behind it) works, you can refresh only a whole table (or partitions of a table, which I cover in [Chapter 16](#)). A refresh operation triggers a full load of the whole content of the table, and the refresh operation doesn't allow you to access the previously stored data. Therefore, you can't implement slowly changing dimensions with Power Query, but need a data warehouse layer (where you can not only store the versions permanently, but where you are also able to update existing rows). In [Part V](#), you will learn about the concept of a data warehouse layer in general, and in ["Slowly Changing Dimensions" on page 387](#), you'll learn how to implement different types of slowly changing dimensions in a relational database.

Usually, an implementation of slowly changing dimensions does not mean any extra effort in the world of Power Query, as the rows in the data warehouse's fact table(s) are already referencing the right version of the dimension tables.

Power BI is special when it comes to hierarchies, as you need to deserialize hierarchies in a way that each level of the hierarchy is represented by one column in the table. The next section has you covered there.

Hierarchies

If you're following along sequentially in this book, you should have denormalized all natural hierarchies in the dimension tables (see [“Denormalizing” on page 270](#)). With the natural hierarchy denormalized, you have all levels of the hierarchy as columns in one single table. Adding them to a hierarchy in Power BI's data model is very easy.

In this section, I concentrate on parent-child hierarchies. They're very common, and you also need to store the names of all parents in dedicated columns, one for each level of the hierarchy. Read on to learn how to achieve this with Power Query.⁶

The solution you need for Power BI is a *materialized path* of the hierarchy. Before you can create the materialized path, you have to merge and expand the query (in my example, it's `DimEmployee`) as many times as levels exist in this hierarchy. In a parent-child hierarchy, the number of levels is never fixed (that's basically the idea of modeling parent-child hierarchies as a self-joining table). So, unfortunately, you need to do some guessing here: how many levels are currently used? Add a buffer amount (and a note in your “to do” list or calendar to regularly check the number of levels in the table) to avoid being surprised by missing information. You need to repeat similar steps (but not the exact same) per level:

1. Go to Home → Merge Queries → Merge Queries to merge the column `ParentEmployeeKey` of the current query (`Employee (PQ 2)`) with column `EmployeeKey` of (again) `Employee (PQ 2)`. Make sure to choose Left Outer as the Join Kind so you don't lose any rows if there are no child nodes available ([Figure 14-26](#)). The resulting code looks like this:

```
= Table.NestedJoin(Source, {"ParentEmployeeKey"}, Source, {"EmployeeKey"},  
"Level -1", JoinKind.LeftOuter)
```

⁶ The examples in this section use the [Hierarchies.pbix](#) file.

Merge

×

Select a table and matching columns to create a merged table.

Employee (PQ 2)

EmployeeKey	ParentEmployeeKey	EmployeeNationalIDAlternateKey	ParentEmployeeNationalIDAlternateKey
1	18	14417807	null
2	7	253022876	null
3	14	509647174	null
4	3	112457891	null

Employee (PQ 2) (Current)

EmployeeKey	ParentEmployeeKey	EmployeeNationalIDAlternateKey	ParentEmployeeNationalIDAlternateKey
1	18	14417807	null
2	7	253022876	null
3	14	509647174	null
4	3	112457891	null

Join Kind

Left Outer (all from first, matching from second)

☐ Use fuzzy matching to perform the merge

> Fuzzy matching options

OK

Cancel

Figure 14-26. Merge query *Employee (PQ 2)* with itself, and make sure to choose the correct columns

- Expand the newly created column and select at least `ParentEmployeeKey` and `FullName`. Make sure to add **Level -1** (or a similar hint) to the column name (Figure 14-27). The resulting code looks like this:

```
= Table.ExpandTableColumn("#Merged Queries", "Level -1",
    {"ParentEmployeeKey", "FullName"}, {"Level -1.ParentEmployeeKey",
    "Level -1.FullName"})
```

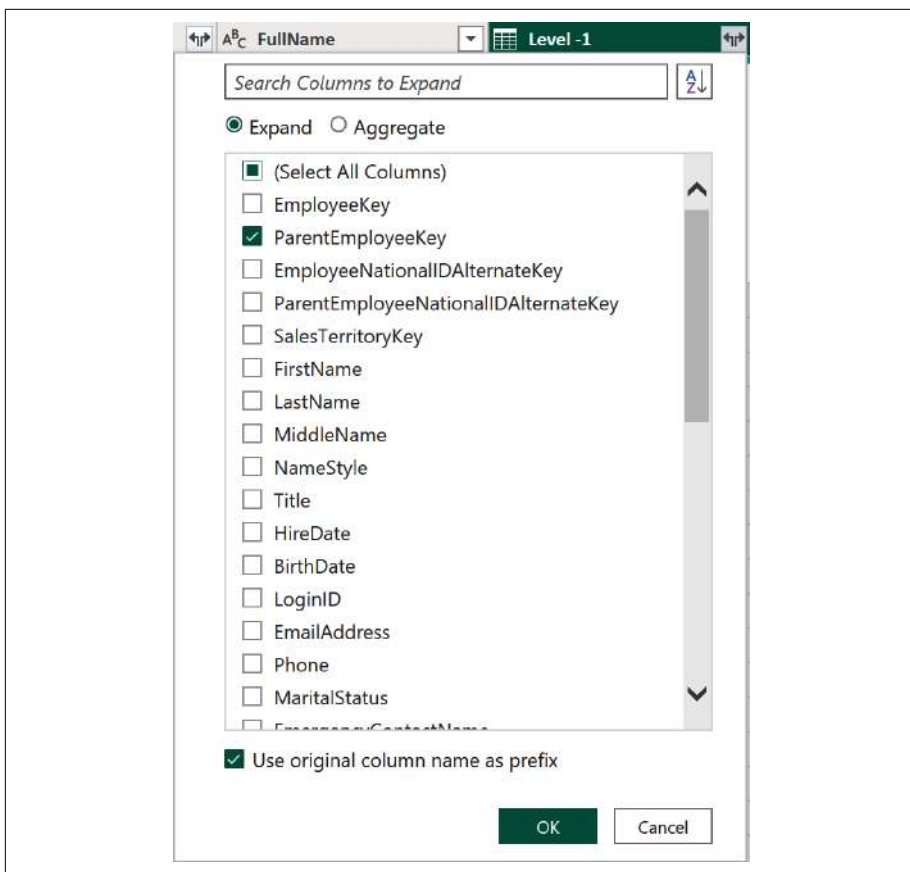


Figure 14-27. Expand the next level

3. Go to Home → Merge Queries → Merge Queries to merge column **Level-1.ParentEmployeeKey** of the current query (Employee (PQ 2)) with column EmployeeKey of (again) Employee (PQ 2). Make sure to choose Left Outer as the Join Kind so you are not losing any rows, in case there are no child nodes available:

```
= Table.NestedJoin("#Expanded Level -1", {"Level -1.ParentEmployeeKey"},
  Source, {"EmployeeKey"}, "Level -2", JoinKind.LeftOuter)
```

4. Expand the newly created column and select at least ParentEmployeeKey and FullName. Make sure to add **Level -2** (or a similar hint) to the column name:

```
= Table.ExpandTableColumn("#Merged Queries -2", "Level -2",
  {"ParentEmployeeKey", "FullName"}, {"Level -2.ParentEmployeeKey",
  "Level -2.FullName"})
```

5. Repeat similarly for all necessary levels.

After you have the `EmployeeKey` and `FullName` for every level as individual columns for every row, you can create the materialized path. Remember, the materialized path is a concatenated list of all the keys of all levels above a node. In Power Query's UI, you simply Ctrl-click all columns containing the keys; e.g., for the current example:

- Level -5.ParentEmployeeKey
- Level -4.ParentEmployeeKey
- Level -3.ParentEmployeeKey
- Level -2.ParentEmployeeKey
- Level -1.ParentEmployeeKey
- ParentEmployeeKey
- EmployeeKey

The order of the keys inside the materialized path is crucial, so pay attention to retaining the order of the columns exactly as described. Then choose **Add Column** → **Merge Columns** from the ribbon (Figure 14-28).

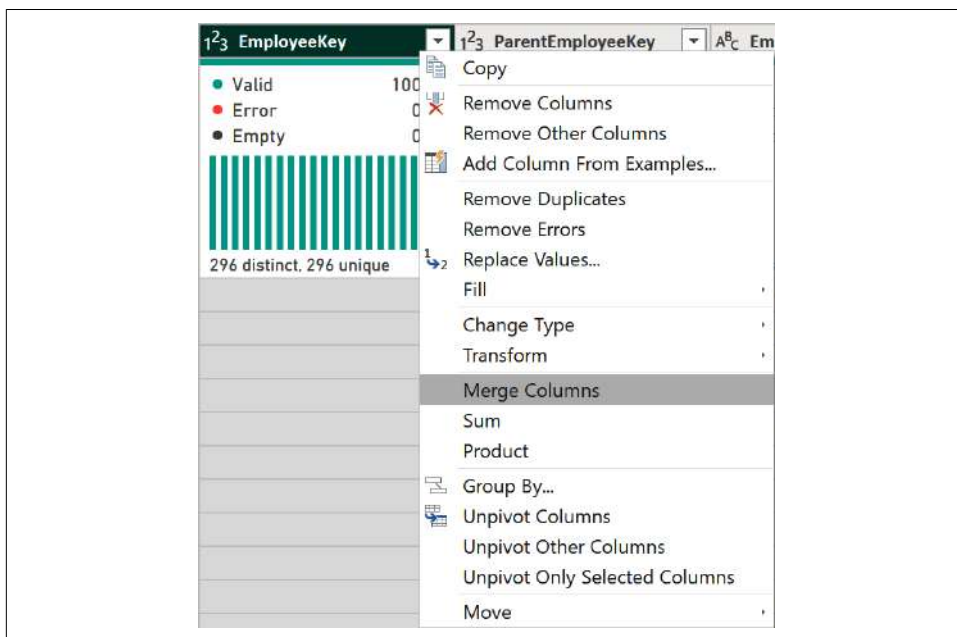


Figure 14-28. Build the Path as a merged columns

The character you choose as the separator isn't so important as long as you make sure that it isn't (and never will be) part of the content of the employee key. I prefer to use the pipe (`|`)—which isn't available in the list. Therefore, I choose "--Custom--" and type in the pipe symbol. As a "New column name (optional)," I provide Path in the dialog box shown in Figure 14-29.

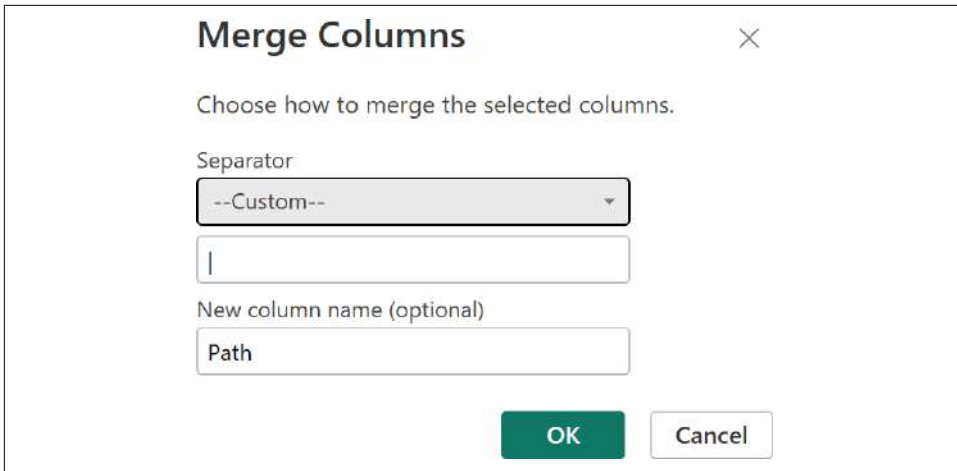


Figure 14-29. Choosing the separator

The resulting code looks like this:

```
= Table.AddColumn("#Expanded Level -5", "Path", each Text.Combine({Text.From(
    ["Level -5.ParentEmployeeKey"]), Text.From(["Level -4.ParentEmployeeKey"]),
    Text.From(["Level -3.ParentEmployeeKey"]), Text.From(
    ["Level -2.ParentEmployeeKey"]), Text.From(["Level -1.ParentEmployeeKey"]),
    Text.From([ParentEmployeeKey]), Text.From([EmployeeKey])}, "|"), type text)
```

With the (materialized) Path at hand, it's now easy to calculate on which level a certain node in the hierarchy is. Just count the separators (|) and add one. Choose Add Column → Custom Column from the ribbon and fill in PathLength for the “New column name” and List.Count(Text.PositionOf([Path], "|", Occurrence.All)) + 1 for the “Custom column formula” in the text box that appears (see Figure 14-30).

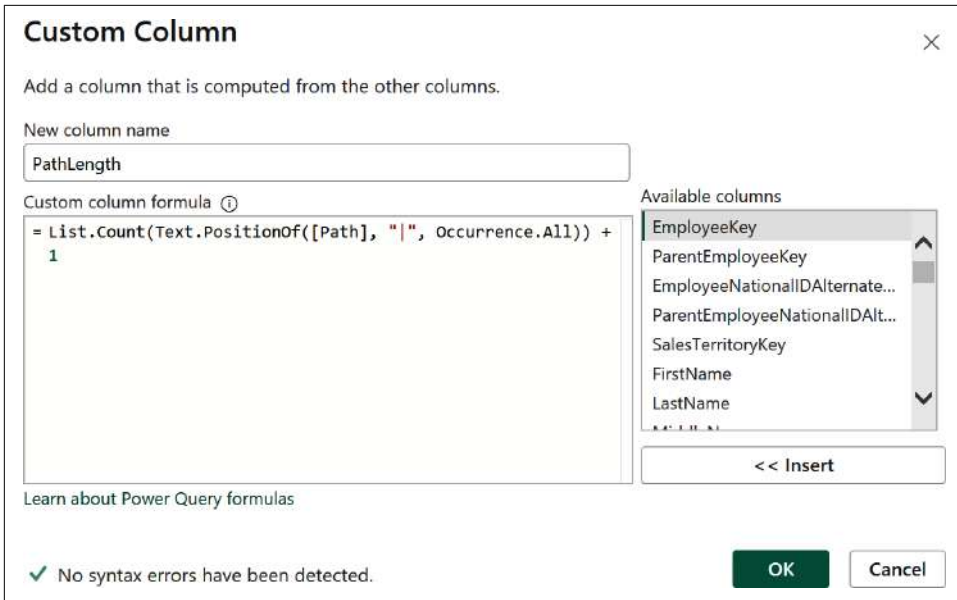


Figure 14-30. Calculating the *PathLength*

It's common to speak of the *leaves* of the hierarchy when talking about the nodes on the very bottom of the hierarchy. *Leaf nodes* are nodes that aren't parents. To find out if a node is a leaf, you need to find out if you can find the node's `EmployeeKey` inside the list of all `ParentEmployeeKeys`. If we find it, it is not a leaf. If we do not find it, it is a leaf. Choose `Add Column` → `Custom Column` from the ribbon once more and fill in "IsLeaf" for the "New column name" and not `List.Contains(Table.Column("#Changed Type", "ParentEmployeeKey"), [EmployeeKey])` for the "Custom column formula" in the text box that appears (see Figure 14-31).

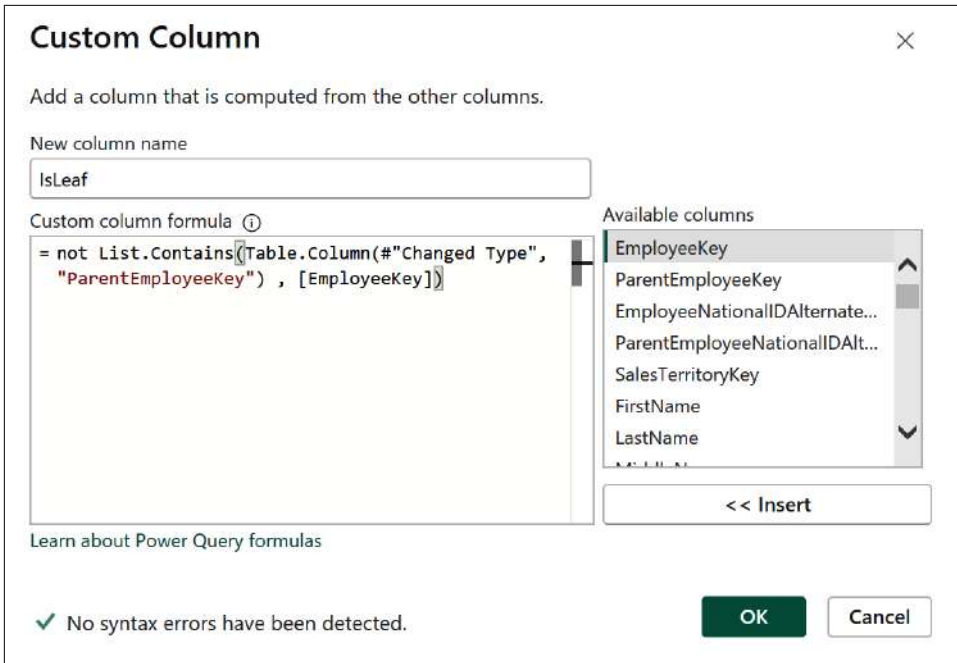


Figure 14-31. Finding out if an employee is on the leaf level of the hierarchy

Finally, it's time for the most important part: creating a column for every level, showing all the names of all the parents of a node.

Unfortunately, the `Level -X.FullName` columns are just in the wrong order (that's why I named them "minus X"). As the hierarchy can be ragged (i.e., not every part of the hierarchy has the same amount of level), you need to start from the top node (e.g., the CEO) and work toward the leaves. The first step is similar to the creation of column `Path`, but it's based on `FullName` (and not `ParentEmployeeKey`). Again, double-check the sequence order of the columns (starting with the column with "-5" in the name and ending with column named `FullName`), as shown in Figure 14-32.

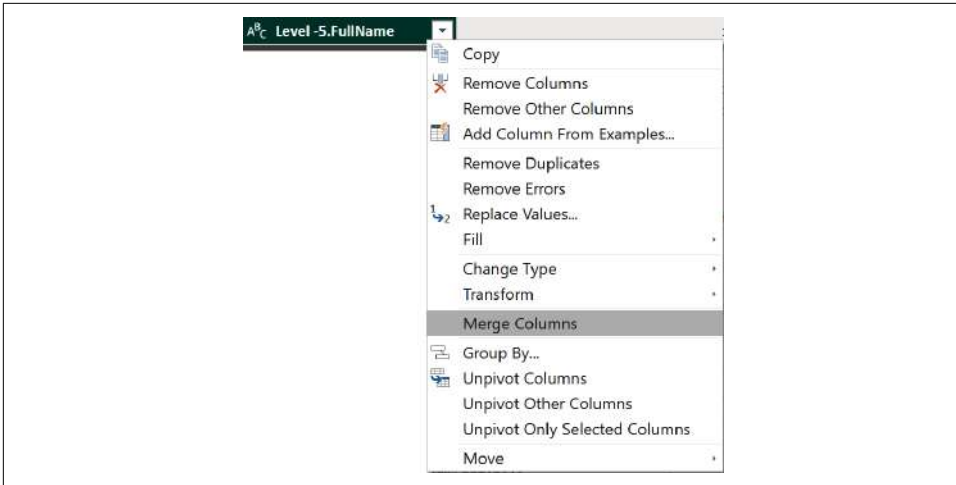


Figure 14-32. Merging the name columns into one

Then you specify the separator and a new name (Figure 14-33).

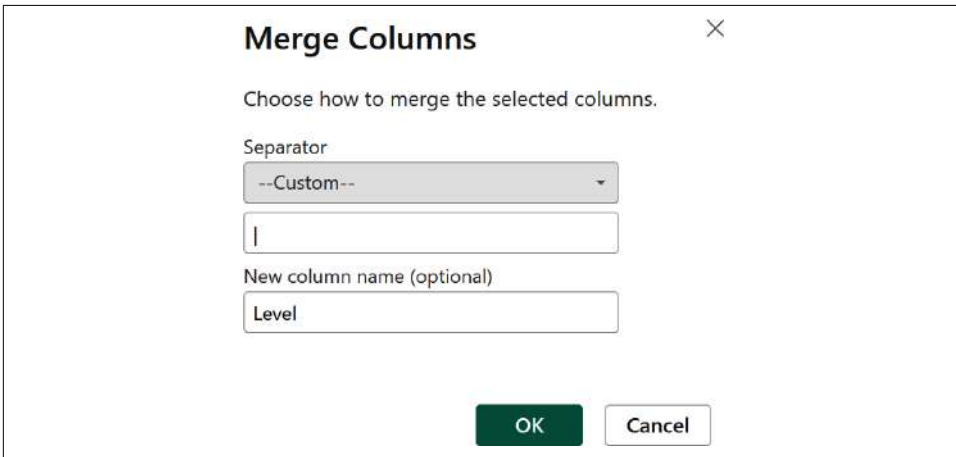


Figure 14-33. Specifying the separator and a new column name

And here is the resulting M code:

```
= Table.AddColumn("#Changed Type1", "Level", each Text.Combine({Text.From(
  ["Level -5.FullName"]), Text.From(["Level -4.FullName"]),
  Text.From(["Level -3.FullName"]), Text.From(["Level -2.FullName"]),
  Text.From(["Level -1.FullName"]), Text.From([FullName])}, "|"), type text)
```

This step was just an intermediate step. By selecting Transform → Split Column → By Delimiter and providing the separator from the step before (|), you can easily split the content of the Level column into individual columns per level (Figure 14-34).

Figure 14-34. Splitting the merged column

Here I have the M code for you:

```
= Table.SplitColumn("#Added [Level]", "Level", Splitter.SplitTextByDelimiter(
    "|", QuoteStyle.Csv), {"Level 1", "Level 2", "Level 3", "Level 4", "Level 5"})
```

This code is not the one that was created by the UI. I removed the dots (.) in the list of names for the level (e.g., I changed “Level.1” to “Level 1” for the sake of better readability).

If you think all these steps are a bit tedious and that guessing about the number of levels isn’t a good approach for a resilient data model, I fully agree. Imke Feldmann developed a function (written in M code) to dynamically dissolve the levels of the parent-child hierarchy and flatten them out into individual columns (as you did manually). [Her post](#) informed the reasoning and steps of this section—I tweaked Imke’s code in the example file to conform to my column and table names and my data model.

Key Takeaways

You learned that Power Query is a powerful tool to achieve all sorts of transformations. I demonstrated many functionalities in the UI and also made sure that you familiarized yourself with the M language, as it can help to make the steps resilient against changes in the data source. I limited myself to what I consider the most important features:

- Normalizing your fact tables involves steps to find candidates for dimensions. Power Query's column quality and column distribution give you an idea of the cardinality of a column, even before you load it into the data model. Always double-check the transitive dependencies to make the right decision.
- You create a dimension table by referencing it, selecting all necessary columns (and removing all others), and remove all duplicates.
- To denormalize a table, you add the information from the related table by merging it into the existing one, and expand the result to add the columns.
- It's important to disable "Enable load" for intermediate queries, which should not be part of the final data model.
- Don't spend too much time on calculations in Power Query. DAX is usually the better place.
- Physically adding variations for a table (for role-playing purposes) is very easy in Power Query. You just reference the original table and add a suffix to all column names, indicating the role.
- Slowly changing dimensions must be solved in a data warehouse layer, which allows you to compare the existing rows with the newly delivered rows in order to insert these rows or update the already existing rows. This is not possible in Power Query.
- You can flatten parent-child hierarchies by applying several steps. Via a function, you can apply these steps in a dynamic way.

In the next chapter, you will learn about more advanced challenges and how to solve them in Power Query.

Real-World Examples Using Power Query and M

In addition to the standard tasks to transform a given data model into a star schema, there is a lot we can do in Power Query to prepare data for advanced challenges. In many cases, Power Query is better than DAX when it comes to shaping data models. Keep in mind, though, that some solutions require DAX measures to be written, which can't be replaced by even sophisticated Power Query or M. I also use scripts in M to make solutions more dynamic, lessening the effort needed to maintain solutions when there are changes in the data source.

The use cases in this chapter are listed here:

- How to group values into bins or buckets to show the name of the bucket instead of the actual value.
- How to support multi-fact data models by bridging the many-to-many relationship, which will appear between some of the relationships between a fact and a dimension table. I will demonstrate the solution on the example of a budget.
- In my multi-language solution, Power Query synthesizes the translations. I will show you how you can use Azure Cognitive Services to get the texts translated.
- Key-value pair tables must be pivoted to be able to satisfy common reporting requirements. You will learn which buttons to click in the UI to pivot the table so that every key becomes a column of its own. I will also show you how you can implement a dynamic solution that will take automatically care of new keys.

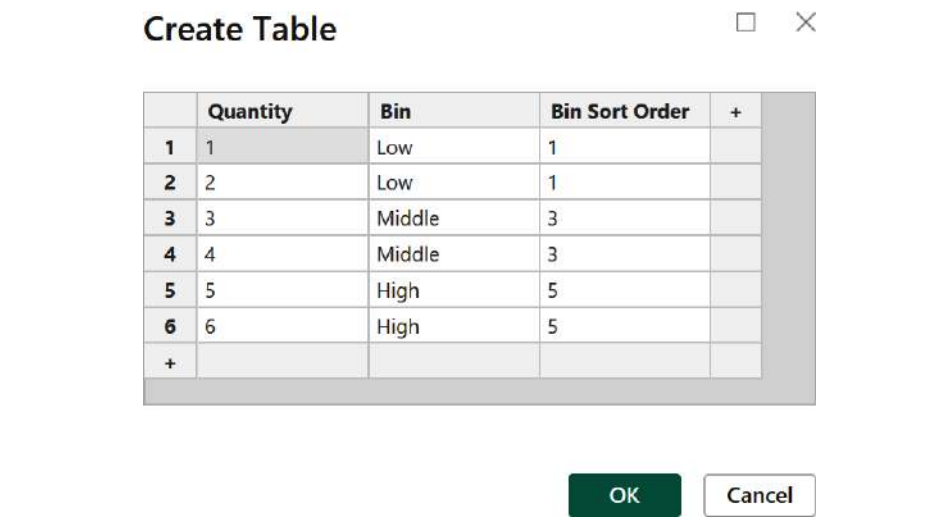
Let's start with binning.

Binning

Let's take a look at how to create either of the two lookup tables from “Binning” on page 58. One of the lookup tables described there was the full list of possible values including a column with the correct bin.¹

Create a Bin Table by Hand

Of course, you can always create a new table by using Home → Enter Data and manually inserting the necessary values (or you can copy and paste from an Excel spreadsheet). You can see such a table in Figure 15-1.



	Quantity	Bin	Bin Sort Order	+
1	1	Low	1	
2	2	Low	1	
3	3	Middle	3	
4	4	Middle	3	
5	5	High	5	
6	6	High	5	
+				

OK Cancel

Figure 15-1. A table containing a row for each quantity

The M script for the table's content is impossible to maintain directly; it's Base64-encoded:

```
= Table.FromRows(Json.Document(Binary.Decompress(Binary.FromText(
    "i45WMLTSUfLJLweShkqx0tFKRmh8YyDLNzMLJScVyDAGC5lgCpkCWR6Z6RLAyhQsYIYiEAsA",
    BinaryEncoding.Base64), Compression.Deflate)), let _t = ((type nullable text)
    meta [Serialized.Text = true]) in type table [Quantity = _t, Bin = _t,
    #"Bin Sort Order" = _t])
```

Instead of the values you entered into the table, you see a step containing a long chain of letters and numbers, which seem to have nothing to do with your input because the values are Base64-encoded. Luckily, you can edit the data of such a table from the

¹ The file for the examples in this section is *Binning.pbix*.

gear icon of the Source step. Resist adding “Replace values” steps to correct typos in the data created via Enter Data and edit the content of the table directly instead.

I like to use this feature (directly entering the data for a table) for quick demos or proofs-of-concept. But I don’t like it for solutions rolled out into production. Instead, I find it way better to create the tables with a script (instead of maintaining the tables themselves).

Deriving the Bin Table from the Facts

As an alternative to the hardcoded table, you can use two other approaches to define the list of values (which are then assigned to a bin). One method involves providing a table containing the ranges per bin (see [“Create a Bin Range Table in M” on page 307](#)). Here, I describe how to reference the fact table to derive a bin table.

Right-click the fact table in the Queries list and choose Reference. Then remove all columns except the one for which you want to create the bins (e.g., Quantity) by right-clicking the column header as shown in [Figure 15-2](#).

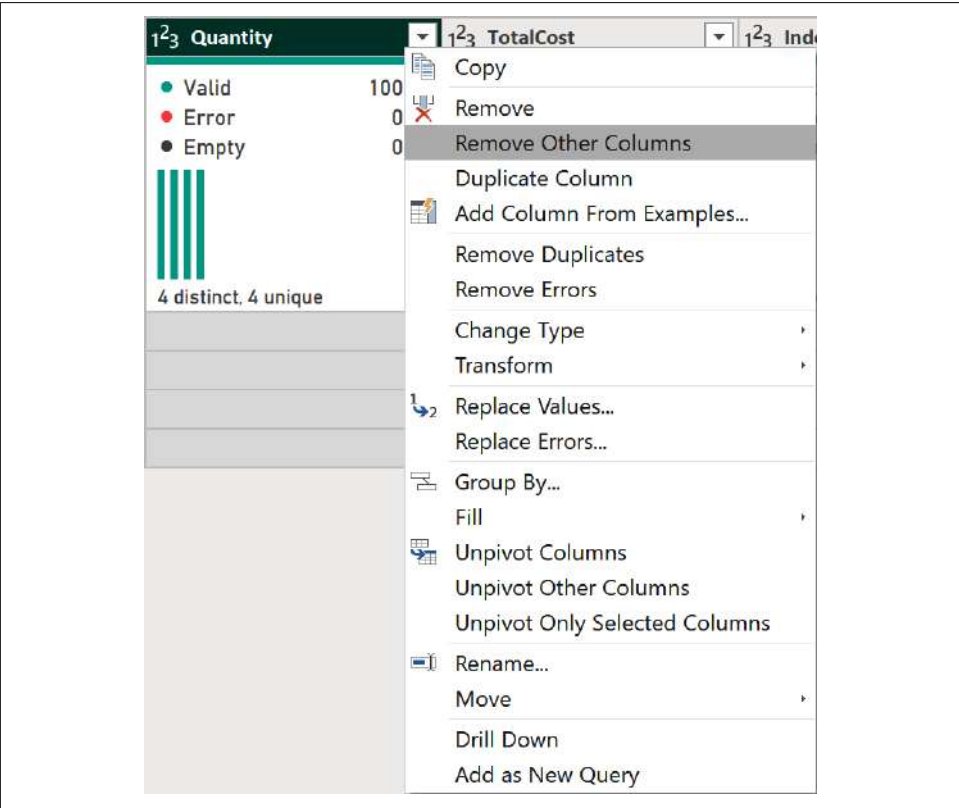


Figure 15-2. Removing all columns except the one that should be binned

Then, remove all duplicates, again via the context menu of the column's header (Figure 15-3).

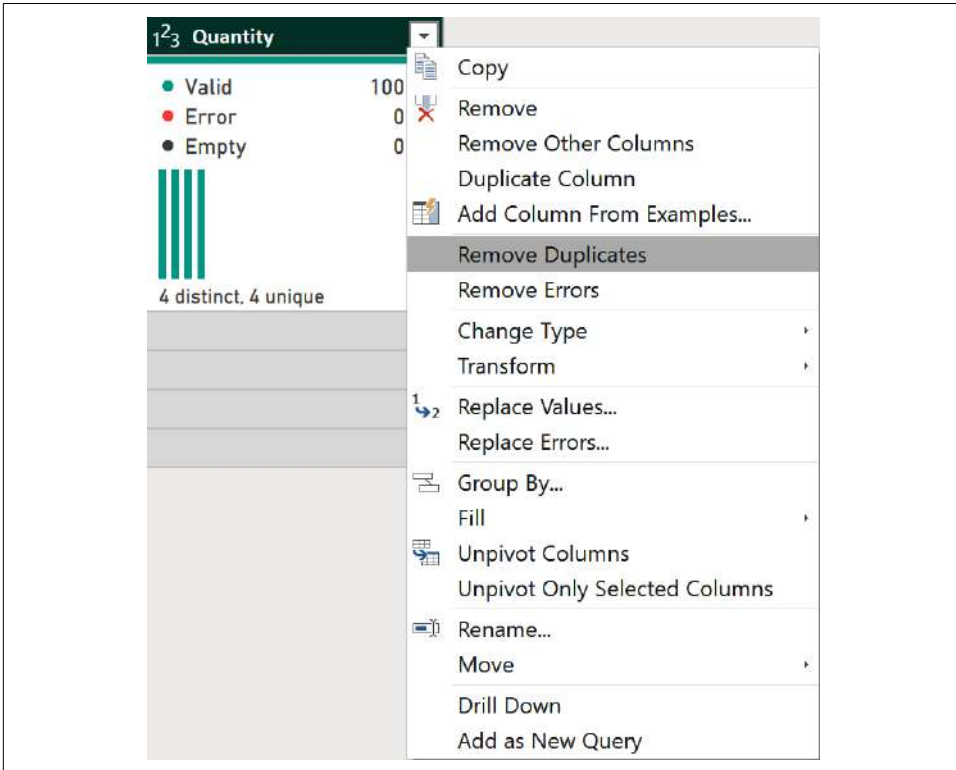


Figure 15-3. Removing duplicates

The generated code looks like this:

```
let
    Source = Sales,
    #"Removed Other Columns" = Table.SelectColumns(Source,{"Quantity"}),
    #"Removed Duplicates" = Table.Distinct(#"Removed Other Columns"),
    ...
```

The result may not have a row for each quantity, but it'll always contain all necessary lookup values used in the fact table. The order of the rows in this table looks a bit unusual—it's not ordered by quantity but by order of appearance of a quantity in the fact table; this isn't a problem.

Before you complete the table, take a look at how to create it with M code.

Create a Bin Table in M

Depending on the size of your fact table, building it can take a while. The alternative is to use the function `List.Numbers` and provide a start and an end value (e.g., Power Query parameters `MinBin` and `MaxBin`; I explain how to create and maintain parameters later). I then make sure to convert this list into a table, give column `Column1` a more descriptive name (e.g., `Quantity`) and the proper data type. This can't be started via the UI but only by creating a new empty query where you call the function and pass in the two parameters:

```
let
    Source = List.Numbers(MinBin, MaxBin),
    ...
```

Then you need to convert the list (returned by function `List.Numbers`) into a table via `Transform` → `To Table` (Figure 15-4).

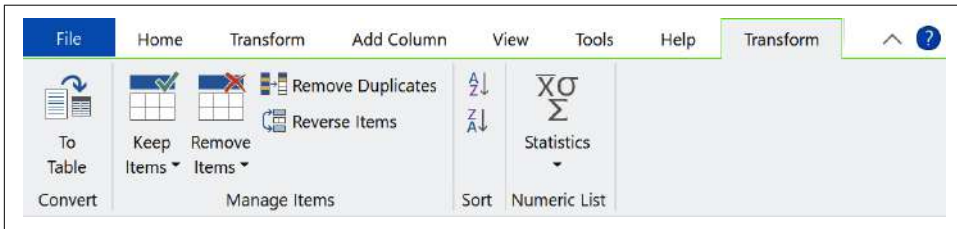


Figure 15-4. Transforming a list into a table

Then I renamed `Column 1` to `Quantity` and chose `Whole number` as the data type in the UI. Here is the first part of this script:

```
let
    Source = List.Numbers(MinBin, MaxBin),
    #"Converted to Table" = Table.FromList(Source, Splitter.SplitByNothing(),
        null, null, ExtraValues.Error),
    #"Renamed Columns" = Table.RenameColumns(#"Converted to Table",{{"Column1",
        "Quantity"}}),
    #"Changed Type" = Table.TransformColumnTypes(#"Renamed Columns",
        {{"Quantity", Int64.Type}}),
    ...
```

In the next step, I add a custom column (`Bin`) containing the bin name per quantity. To decide which bin a value falls into, I provide *parameters* (which are easy to spot and understand) and use the parameter values. If I need to change the borders of the bins, I don't need to scan through all the applied steps of all the queries but can simply change the parameter value.

Figure 15-5 shows the parameters I have created for the purpose of this demonstration. `MediumBin` is selected. Via `Home` → `Manage Parameters` → `Manage Parameters`,

you can create *New* parameters, change the settings for an existing parameter, or remove it via the X icon next to the parameter's name.

The screenshot shows the 'Manage Parameters' dialog box. On the left, a list of parameters is displayed: PathAndFile, MinBin, MediumBin (highlighted), HighBin, MaxBin, ServerName, and DatabaseName. Each parameter has a small icon to its left. A 'New' button is at the top of the list. To the right of the list, the configuration for the selected 'MediumBin' parameter is shown. It includes a 'Name' field with 'MediumBin', a 'Description' field with 'Boundary, where the "Medium" bin begins', a 'Required' checkbox that is checked, a 'Type' dropdown menu set to 'Decimal Number', a 'Suggested Values' dropdown menu set to 'Any value', and a 'Current Value' field with '3'. At the bottom right, there are 'OK' and 'Cancel' buttons.

Figure 15-5. Managing the properties of a Power Query parameter

A Power Query parameter has the following properties:

- A mandatory Name.
- An optional Description, which is shown as a tooltip if you mouse over to select the parameter name in the list of queries.
- A checkbox to define if a value for this parameter is Required.
- A data Type chosen from the list of available values. The available data types match the data types available for a column. You can find a complete list and explanation in [“Tables or Queries” on page 244](#).



Choose the data type of each parameter wisely. Leaving the data type at the default value Any is a common mistake that will prohibit changing the parameter's value in the Power BI service. Other problems can occur if you don't choose the right Date/Time-related data type (e.g., unexpected behavior from a filter based on the parameter).

- The ability to decide, through Suggested Values, whether the parameter's value can be any value and changed via a simple input field, or if it needs to come from a hardcoded list of values, or a value listed in a Power Query query.
- A parameter currently set to *Current Value*. This value is also saved in the .pbix file and will be used for all future refreshes, unless changed here or in the Power BI service.

Let's use these parameters in a script to add the name of the bin. Choose Add Column → Custom Column from the ribbon (Figure 15-6). In the dialog box, you can then add the expression to return Low, Middle, or High according to the bin range table.

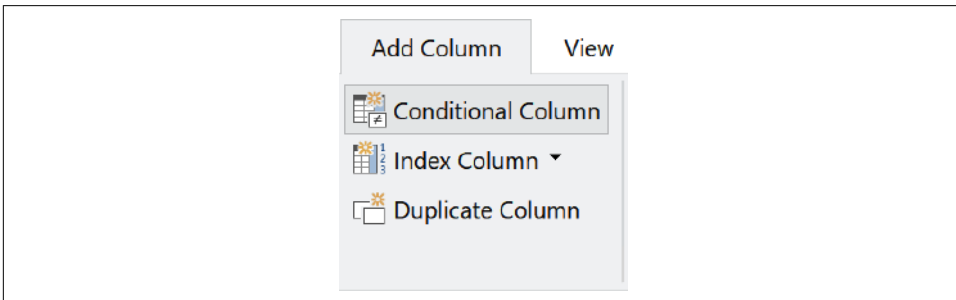


Figure 15-6. Adding a custom column

Here is the full M code:

```
= Table.AddColumn(#"Changed Type", "Bin", each
    if [Quantity] < MediumBin then "Low"
    else if [Quantity] < HighBin then "Middle"
    else "High")
```

The code for the column Bin uses if then - else if then chains (due to the lack of SWITCH or CASE keywords in Power Query). I didn't hardcode the borders of the bins but referenced Power Query parameters MediumBin and HighBin to decide which bin a value should fall into. If these boundaries change over time, nobody needs to touch the code—only changes to the content of the parameters would be required. And only if the number of bins increases do we need to add parameters and a new else if then line to the step where we define the custom column. You can

achieve the same result by using the GUI and selecting Add Column → Conditional Column from the ribbon (see [Figure 15-7](#)).

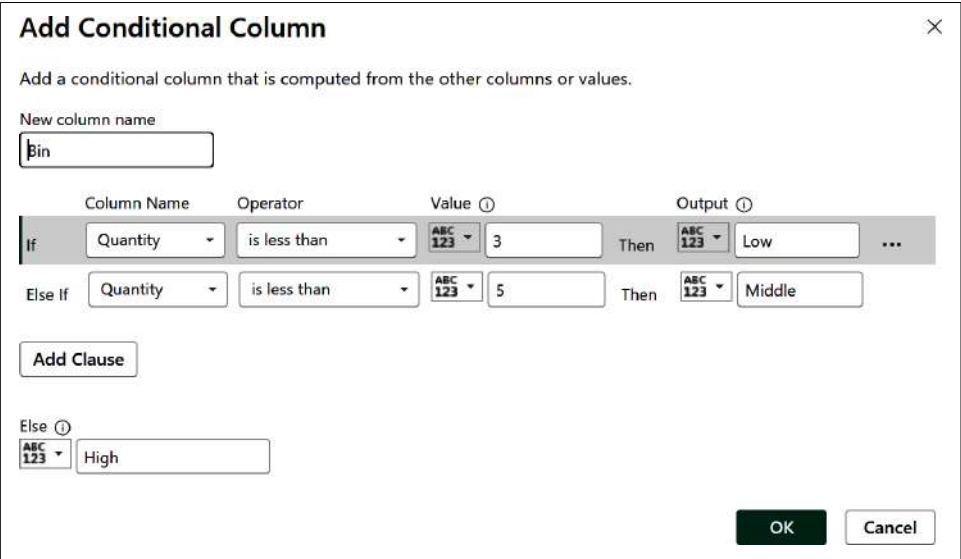


Figure 15-7. Specifying the condition to return Low, Middle, or High as the bin’s name

I add another column (`_SortOrder`) to be able to order the bin names in a custom order (Low first, then Middle, and High last) instead of alphabetical order (which would be High, Low, and Middle and confuse most report users). For this column, I implement a logic to pick the smallest quantity of each bin. For this to happen, I first group the result of the previous step (Added Custom) by bin name. Choose Transform → Group by in the ribbon ([Figure 15-8](#)).

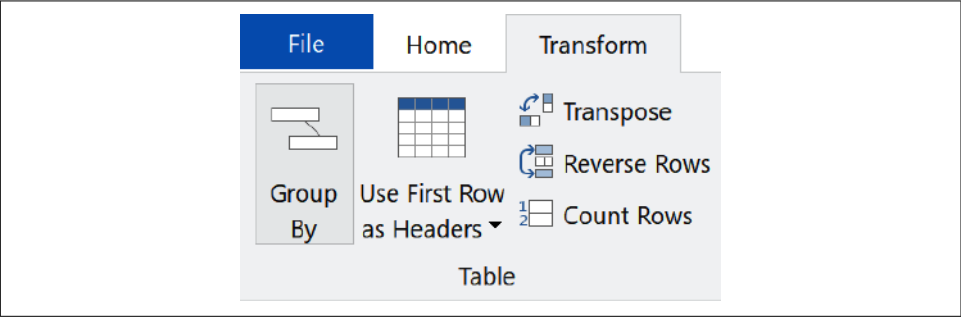
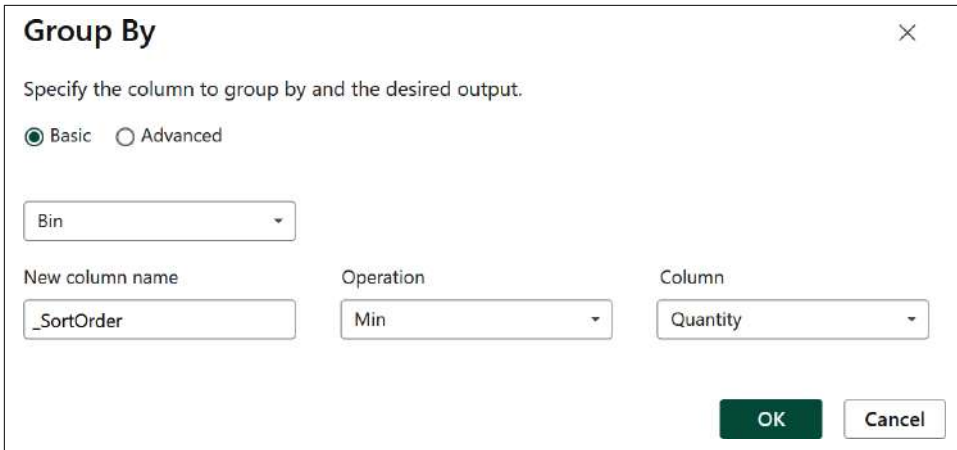


Figure 15-8. Grouping a query

Specify the `Bin` column in the drop box, enter `SortOrder` for the New Column Name, chose *Min* as the *Operation* (to get the minimal quantity value per bin as its sorting value), and choose *Quantity* from the *Column* drop box ([Figure 15-9](#)).



Group By [Close]

Specify the column to group by and the desired output.

☒ Basic ☐ Advanced

Bin

New column name: _SortOrder Operation: Min Column: Quantity

[OK] [Cancel]

Figure 15-9. Specifying the grouping

I then merge the result of the Grouped Rows step with with Added Custom step. Here, you need to right-click the last step and choose Insert Step After (Figure 15-10).

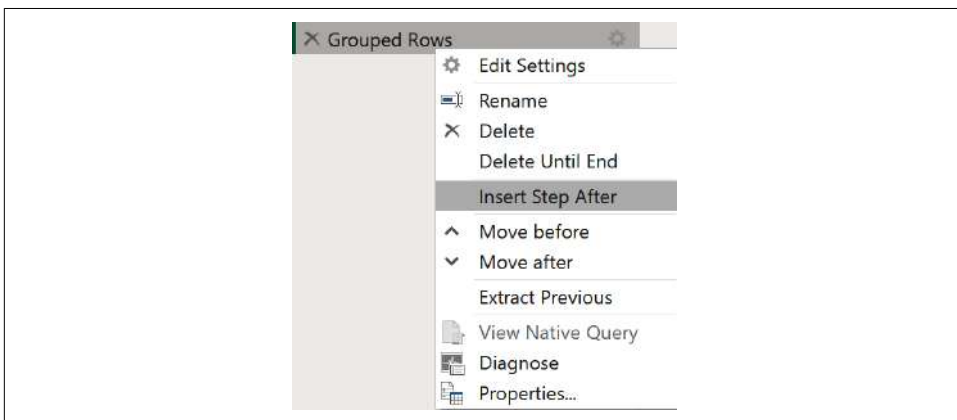


Figure 15-10. Insert a step after Grouped Rows

Insert the following code:

```
= Table.NestedJoin("#Added Custom", {"Bin"}, #"Grouped Rows", {"Bin"},
    "Grouped Rows", JoinKind.LeftOuter)
```

Finally, you need to expand the _SortOrder column. Click the Expand icon to the left of the column name in the column header (Figure 15-11).

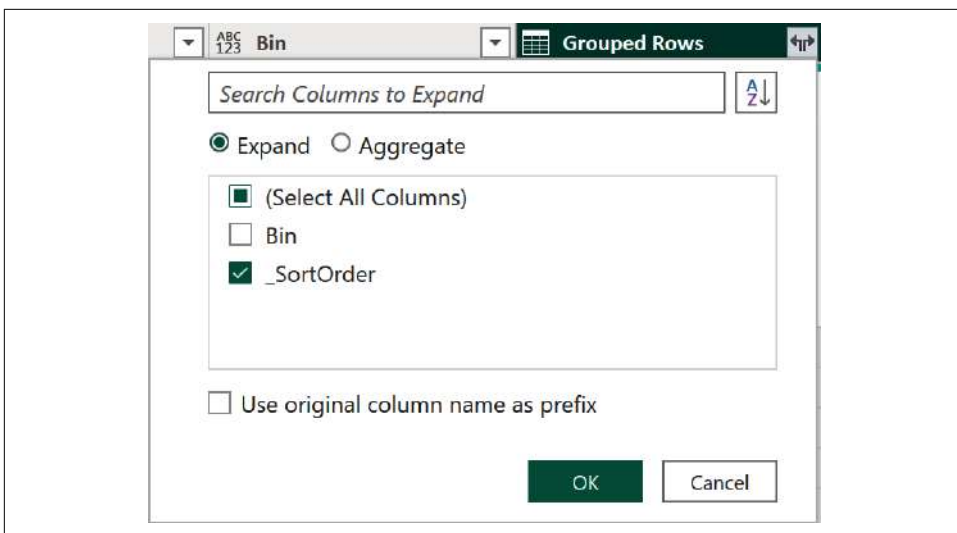


Figure 15-11. Expanding the `_SortOrder` column

Here's the code for the last three steps:

```
...
#"Grouped Rows" = Table.Group(#"Added Custom", {"Bin"}, {{"_SortOrder",
    each List.Min([Quantity]), type nullable number}}),
#"Merged Queries" = Table.NestedJoin(#"Added Custom", {"Bin"}, #"Grouped Rows",
    {"Bin"}, "Grouped Rows", JoinKind.LeftOuter),
#"Expanded Grouped Rows" = Table.ExpandTableColumn(#"Merged Queries",
    "Grouped Rows", {"_SortOrder"}, {"_SortOrder"})
```



Don't forget to provide column `_SortOrder` as the “Sort by column” for column `Bin` in Power BI's “Column tools.”

In [Table 15-1](#), you see the final result for the approach based on the fact table, which lists only quantities existing in the fact table in the order of appearance of a quantity in the fact table.

Table 15-1. A bin table derived from the values available in the fact table

Quantity	Bin	_SortOrder
3	Middle	3
4	Middle	3
1	Low	1
5	High	5

Table 15-2 shows the final result for the approach based on the `List.Numbers` function, which is a complete list of values, independent of whether a value appears in the fact table.

Table 15-2. A bin table derived from a list of values

Quantity	Bin	_SortOrder
1	Low	1
2	Low	1
3	Middle	3
4	Middle	3
5	High	5
6	High	5
7	High	5
8	High	5
9	High	5
10	High	5

Create a Bin Range Table in M

A completely different approach to model binning in Power BI, also laid out in “[Binning](#)” on page 58, is to provide a table containing the ranges per bin. You can easily create a table in M via the function `#table`:

```
let
    Source = #table(
        type table [#"Low (incl.)" = number, #"High (excl.)" = number,
            #"Bin name" = text],
        {
            {null, MediumBin, "Low"},
            {MediumBin, HighBin, "Medium"},
            {HighBin, null, "High"}
        }
    )
in
    Source
```

The first parameter is optional, but I recommend specifying the names and data types of the columns of the table so you have everything in one place, instead of adding steps afterward.

The second parameter then specifies the content of the table, using the `{}` syntax. You need an outer `{}` for the whole table, and then inner `{}`s per row. Inside the row, provide a comma-separated list of values. Again, I don’t provide the ranges per bin in a hard coded fashion but reference the parameters. In the code, you can see the advantage of having an inclusive lower range and an exclusive higher range: I can

provide parameter `MediumRange` for both the high range of the Low bin and the low range of the Medium bin. In “[Binning](#)” on page 216, I demonstrate how you can use such a table as a lookup table for the bins in your DAX measure.

In my example, such a table looks like [Table 15-3](#).

Table 15-3. A bin table derived from a list of values

Low (incl.)	High (excl.)	Bin name
null	3	Low
3	5	Medium
5	null	High

In the following example, I show how to create a bridge table in Power Query.

Budget

The budget problem is a classic example for a multi-fact data model. You may need more than one fact table if you’re working with content of a different granularity; usually the budget is on a different granularity level than the actual values. I prefer to connect a fact table of different granularity level than the relevant dimension tables is over a bridge table. Such a bridge table is simply a distinct list of the dimensional values on the fact’s granularity level. The fact’s granularity might be either on the dimension’s primary key or a different column.

In the example in “[Budget](#)” on page 60, the fact table (Budget) is not on the same level of granularity as the product table but that of the the product’s product group. The bridge table must be on the same level of granularity as the product group in this case. This table is then “inserted” into the data model between the dimension table and the fact table which has the coarser granularity.²

Creating a bridge table involves the same steps as normalizing tables and creating dimension tables. There are a few differences, though:

- You need to create a distinct list of the common values (e.g., the product group’s name) from both the fact table (Budget) and the dimension table (Product).
- You keep all tables: the base fact table, the base dimension table, and the resulting bridge table.

² These examples use the [Budget.pbix](#) file.

“Normalizing” on page 258 provides a detailed description of the necessary steps to normalize a table. Here, I describe the necessary steps to create the bridge table between the Budget and the Product tables of my example.

First, I referenced query Budget (by right-clicking the query Budget in the query list and selecting Reference) as shown in Figure 15-12 and renamed it to Budget Product Group (PQ).

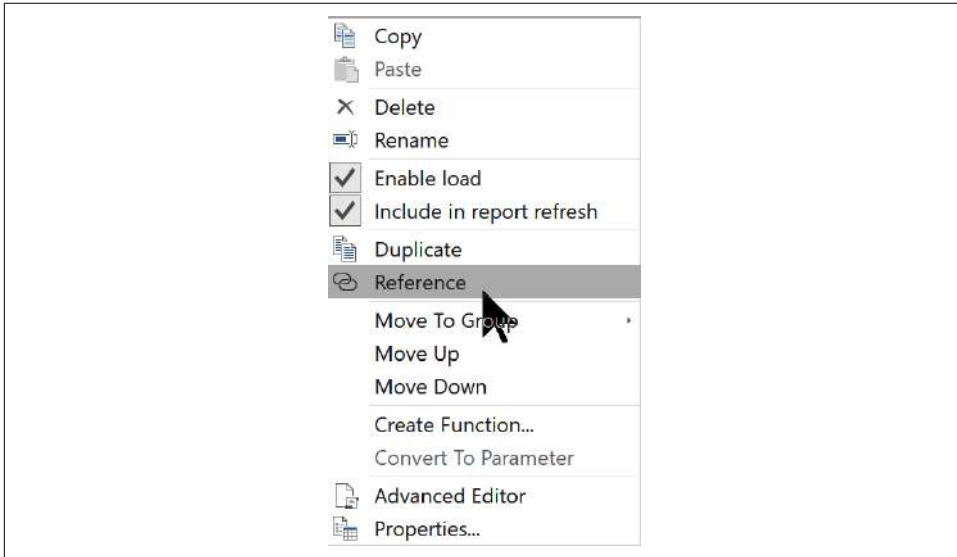


Figure 15-12. Referencing the Budget table

Right-click the newly created query and make sure to disable the “Enable load” option (Figure 15-13). This query will only be an intermediate query, which shouldn’t be loaded into the data model.

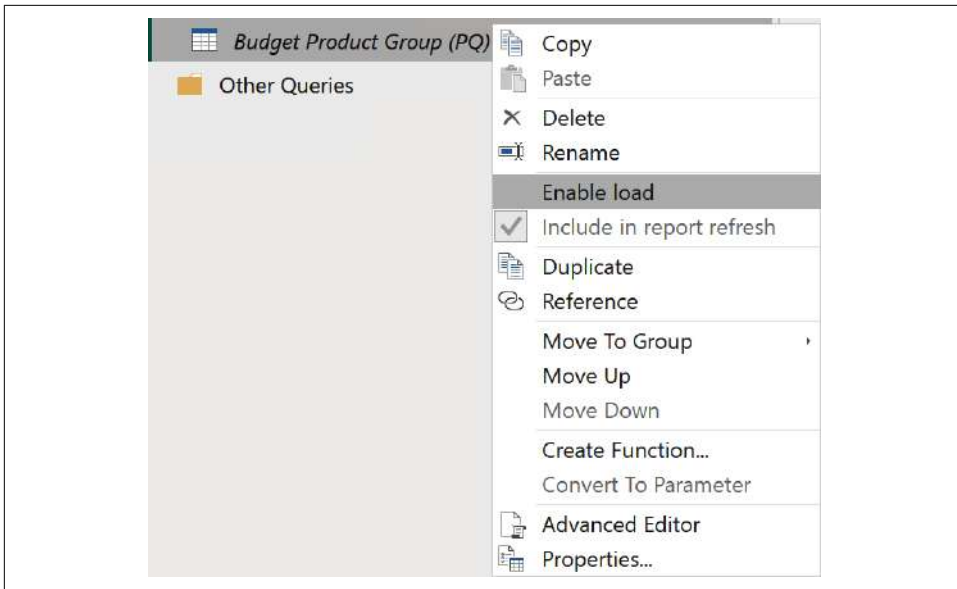


Figure 15-13. Disabling the load for Budget Product Group (PQ)

Then, right-click column Product Group and choose Remove Other Columns to only keep this column, as shown in Figure 15-14.

The code of these steps looks like the following:

```
let
    Source = Budget,
    #"Removed Other Columns" = Table.SelectColumns(Source,{"Product Group"})
in
    #"Removed Other Columns"
```

Next, create the query for the bridge table and load it into the data model. The first steps are similar to the steps for creating the Product Group query: reference query Product (by right-clicking the query Product in the query list and selecting Reference) and rename it Product Group (PQ). Then, right-click the column Product Group and choose “Remove other columns” to keep only this column.

There are two additional steps: Choosing “Append queries” from the Home ribbon and selecting the intermediate query from the Budget Product Group (PQ) steps, appending it to the current one (Figure 15-15).

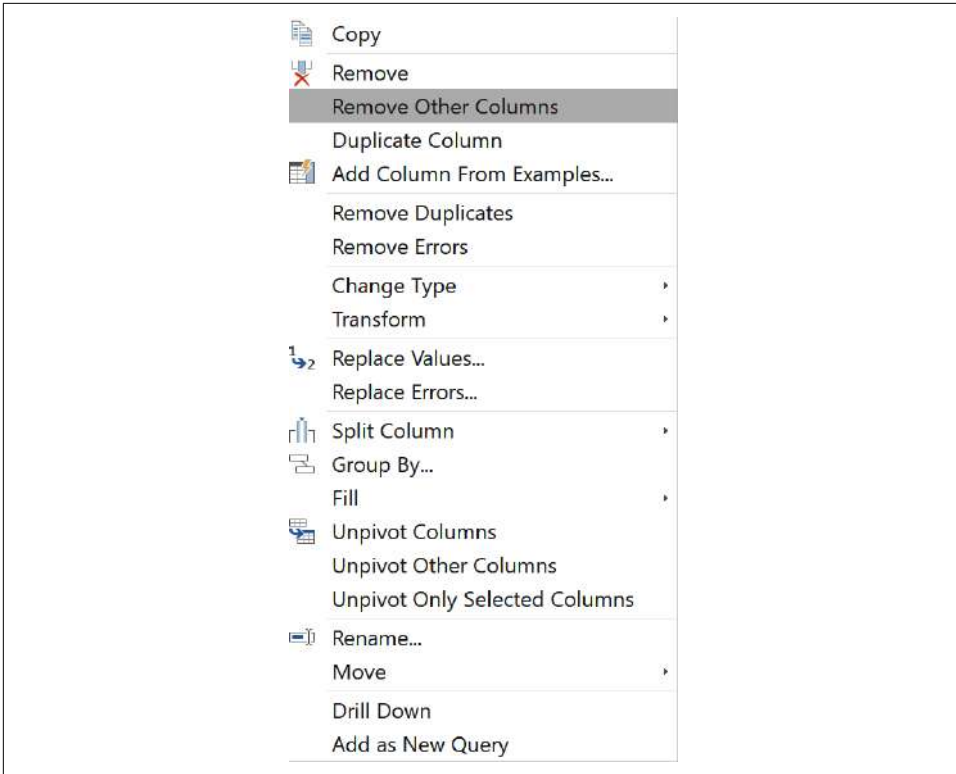


Figure 15-14. Removing columns other than Product Group

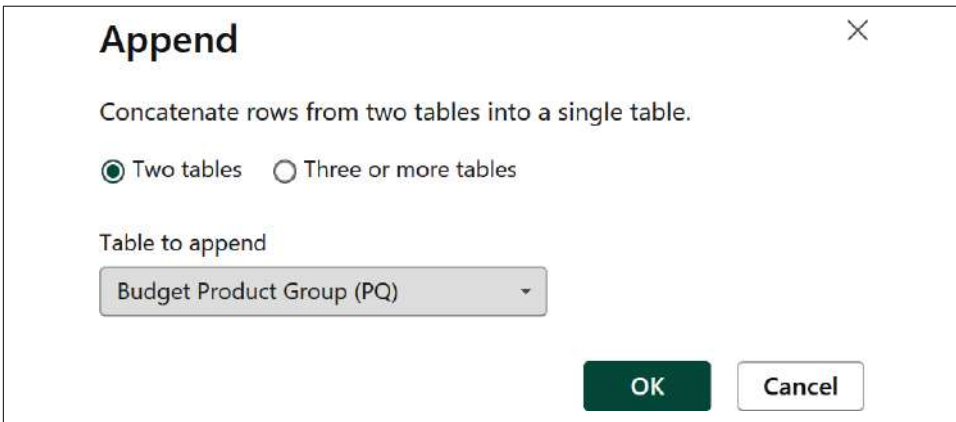


Figure 15-15. Appending query Budget Product Group (PQ) to query Product Group (PQ)

Then, right-click the column header and choose Remove Duplicates (Figure 15-16). This will give you a distinct list of product groups, derived from the Product and Budget tables.

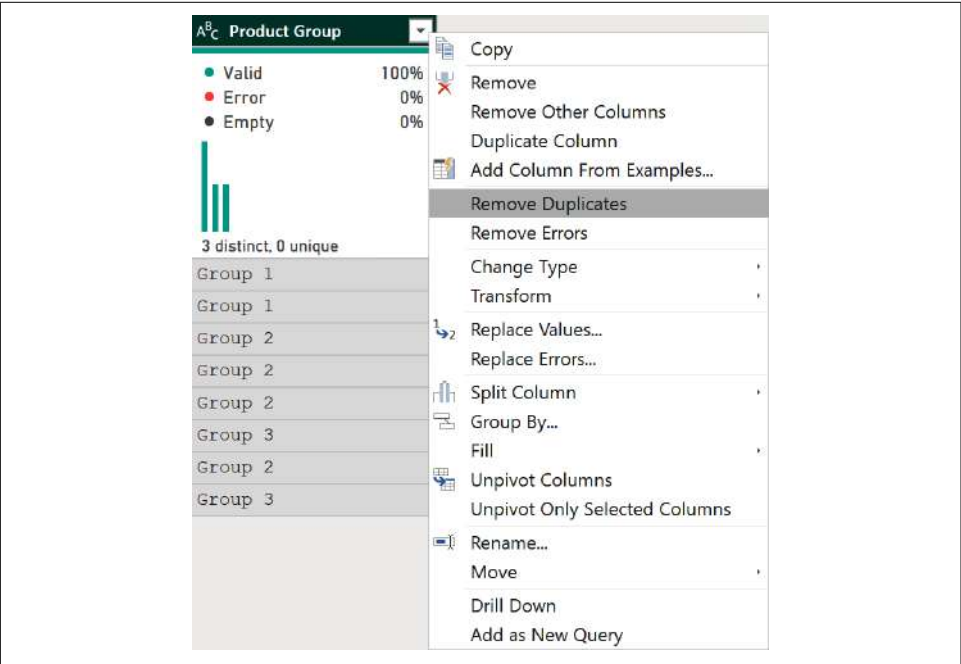


Figure 15-16. Removing duplicate product groups from the query

The code of these steps looks like this:

```
let
    Source = Product,
    #"Removed Other Columns" = Table.SelectColumns(Source,{"Product Group"}),
    #"Appended Query" = Table.Combine({#"Removed Other Columns",
    #"Budget Product Group (PQ)"}),
    #"Removed Duplicates" = Table.Distinct(#"Appended Query")
in
    #"Removed Duplicates"
```

The final result looks like Table 15-4.

Table 15-4. A distinct list of product groups

Product Group
Group 1
Group 2
Group 3

In the next section, I share the steps necessary to implement my concept of a multi-language model in Power Query.

Multi-Language Model

This section concentrates on the `TextComponent` table described in “[Multi-Language Model](#)” on page 139, which contains all the descriptive text for a report (headlines, buttons, etc.). I show how to solve two challenges with the help of Power Query:

- Translating text into different languages
- Pivoting the table to get one column per text prompt and one row per language

Imagine you need to create a report for different languages—including some languages that you don’t understand yourself. Of course, you could hire a translator to do that job. But this chapter is about Power Query, so let’s see how you can use Azure Cognitive Services to translate texts for you.³

Azure Cognitive Services offer several services, including a translations API, which you can call in a step in Power Query. You send text to the API and tell it which language you need. The service will return the text in the chosen language. At the time of writing, the API [supports translations from and to over one hundred languages](#), including Klingon. (Now you know the secret of how I translated the description fields for the sample dimensions into Klingon.)

Before you can start with the API, you need an Azure account with Translator in your subscription. (If you just want to play around with this feature, a free Azure account will be sufficient.) After you create the Translator service in your subscription, you’ll receive an API key.



Treat this key as carefully as you treat your passwords. Everybody who knows this key will be able to use the services at your cost. That’s why I made sure to not expose the full key in the screenshots shown in this section. If you think the key has been leaked, then you should immediately change the key in the service (and at every place where you use the key to connect to the API).

Power Query’s UI doesn’t have a button to connect to the Translator API. (Some Cognitive Services are exposed via the AI Insights ribbon. You need a premium subscription to use them. Translator API isn’t part of the AI Insights offering at the time of writing.) It makes sense to encapsulate the steps to call the API in a Power Query function. Fortunately, the M code isn’t too heavy:

³ The file used for the examples in this section is [Multilanguage.pbix](#).

```

(text, language) =>
let
    body = "[{"Text":""," & text & ""}]",
    jsonContent = Text.ToBinary(body, TextEncoding.Ascii),
    //language = if isempty(language) then "en" else language,
    source= Web.Contents(
        "https://api.cognitive.microsofttranslator.com/translate?api-version=
        3.0&to=" & language,
        [
            Headers=
            [{"Ocp-Apim-Subscription-Key" = apikey_Translation,
            #"Content-Type"="application/json",
            #"Accept"="application/json"}],
            Content=jsonContent
        ]
    ),
    json = Json.Document(source),
    json1 = json{0},
    translations = json1[translations],
    translations1 = translations{0}
in
    translations1

```

The function has two parameters: the text (which should be translated) and the language (into which the text should be translated). It then embeds the text into a string, representing the body for the API call. The body is converted into a binary format (Text.ToBinary). The call to the API is done via the function Web.Contents. The language parameter is appended to the API's URL. Inside the Headers, the Power Query parameter apikey_Translation is referenced (this is the Power Query parameter where you should paste the API key from the the Translator service). The result is stored in the step source, which is then treated as a JSON document, from the first row ({0}) in which the column [translations] is referenced, and again from the first row returned.

Before you can set this function into action, you need to do a cross join between the Textcomponent table and the Language table. **“Multi-Language Model” on page 313** covers how to cross join in Power Query: add a new column that references the Language table, then expand the Language ID:

```

#"Added Language" = Table.AddColumn(#"Changed Type", "Custom", each Language),
#"Expanded Language" = Table.ExpandTableColumn(#"Added Language", "Custom",
    {"Language ID"}, {"Destination Language ID"}),

```

My example texts are available only in English. The cross join will duplicate the existing English text prompts per language available in the Language table. Before calling the Translate API function, add a check: you want to call the API only if the language is not English. If the language is English, add an empty record instead ([]), which avoids calling the API for nothing (and saves money, as you pay per call).

Expanding the new column will contain the translations for all languages, except for English, which will be empty.

In the step `Replace empty translations`, add a `Custom` column containing either the translated text or the original English text. This column is later renamed `DisplayText`:

```
#"Translate API" = Table.AddColumn(#"Expanded Language", "Translate API",
    each if [Language ID]=[Destination Language ID] then []
    else #"Translate API"([DisplayText], [Destination Language ID])),
#"Expanded Translate API" = Table.ExpandRecordColumn(#"Translate API",
    "Translate API", {"text"}, {"DisplayText translated"}),
#"Replace empty translations" = Table.AddColumn(#"Expanded Translate API",
    "Custom", each if [DisplayText translated] = null then [DisplayText]
    else [DisplayText translated]),
```

Now you should have a table with three columns: a technical identifier (Text component), a `Language ID`, and the `DisplayText`. Pivot this result on the `Textcomponent` column, providing `DisplayText` as the Values Column and Minimum as the Aggregate Value Function (hidden under the “Advanced options” in the dialog window).

Voilà, you now have a table with one row per language and one column per text prompt. When a language is selected the column will only show one value. Don’t forget to create an active relationship between this table and the `Language` table in the data model. The cardinality of this relationship will be one-to-one (as the newly created table only contains one column per language).

Of course, you can also easily create bridge tables for all dimension tables with Power Query, as laid out in [Table 11-3](#). Power Query also has nice capabilities to dynamically pivot a key-value pair table, as you will learn in the next section.

Key-Value Pair Tables

In some situations, you will face a table structured like [Table 15-5](#).⁴

Table 15-5. 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

⁴ The examples in this section use the *Key-Value.pbix* file.

ID	Key	Value	Type
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

But you may want (or need) to transform a key-value pair table so you end up with one column per key (instead of one row per key), as shown in [Table 15-6](#).

Table 15-6. The key-value pairs table pivoted on the key column

ID	name	city	revenue	firstPurchase
1	Bill	Seattle	20,000	1980-01-01
2	Jeff	Seattle	19,000	2000-01-01
3	Markus	Alkoven	5	2021-01-01

In this section, I present three solutions to the same problem; each solution is building upon the previous one to make it more resilient to sudden changes in the content of the key-value pair table. Such changes should be expected; the point of such a table is to store data in a flexible fashion. Here are the three ways to get the result of having one column per key with a key-value pair table:

- Using the GUI (over and over again)
- Using M code (and not touching the query again)
- Writing an M function (which can find the data type itself)

Using the GUI

Let's walk through the steps to pivot a table in the GUI.

First, remove column Type because its content isn't needed in the final result, and keeping it would lead to an unwanted result in the next step ([Figure 15-17](#)).

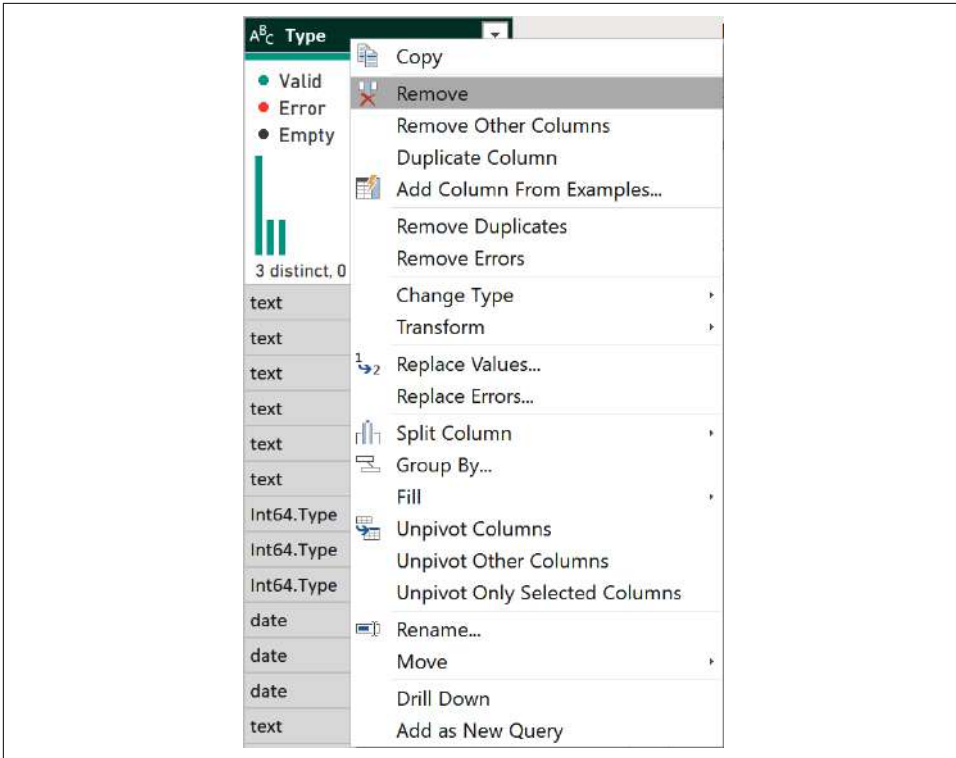


Figure 15-17. Removing the Type column

Then, you need to select the header of column Key and select Transform → Pivot Column from the ribbon (Figure 15-18).

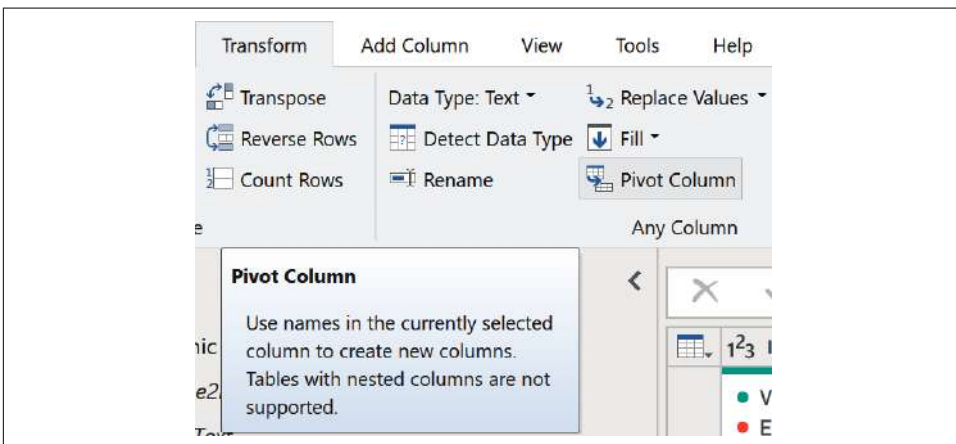


Figure 15-18. Pivoting the Key column

In the dialog box that appears, choose the Value column for Values Column (easy to remember, right?).

It's very important to expand the "Advanced options" and select either Maximum or Minimum as the Aggregate Value Function. This is necessary to have only one value per key. Whether you choose the alphabetically last or first value is of lesser importance. In theory, there should be only one value per Key (and ID), but you still need to tell Power Query what to do if the table contains more than one value for a combination of ID and Key (Figure 15-19).

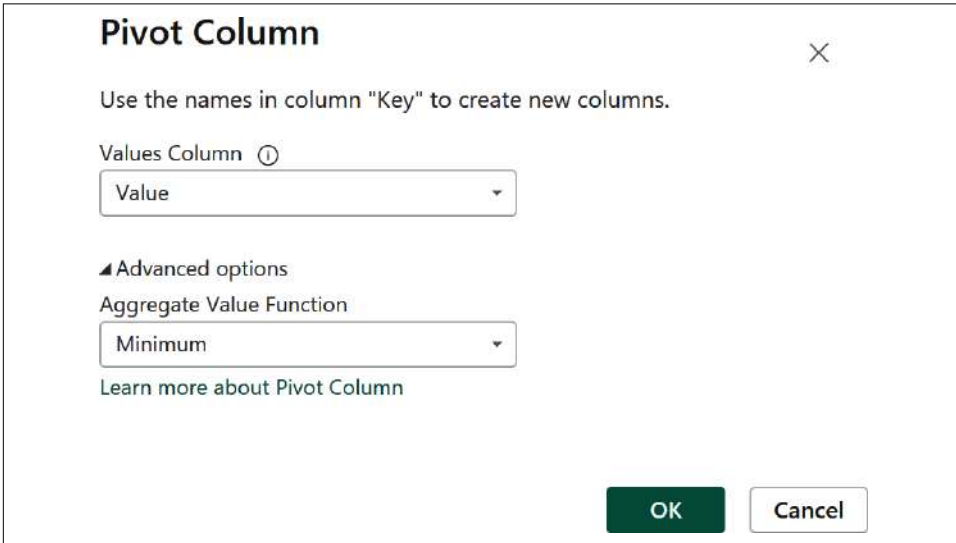


Figure 15-19. Pivoting the Key column



If you choose Don't Aggregate as the Aggregate Value Function, it'll work only as long as there is only one value per key. If you duplicate one of the entries in the example, you'll receive an error: "Expression.Error: There weren't enough elements in the enumeration to complete the operation." I find the message rather confusing, but you can easily fix it by making sure to choose either Maximum or Minimum as the Aggregate Value Function.

The steps so far create the following script:

```
let
    Source = Source,
    #"Removed Columns" = Table.RemoveColumns(Source,{"Type"}),
    #"Changed Type" = Table.TransformColumnTypes(#"Removed Columns",
        {{"ID", Int64.Type}, {"Key", type text}, {"Value", type text}}),
    #"Pivoted Column" = Table.Pivot(#"Changed Type",
        List.Distinct(#"Changed Type"[Key]), "Key", "Value", List.Min),
```


All the newly created columns are now of data type text, because their data type is inherited from column Value, which was of data type text. And column Value is of data type text because that's the common denominator to store information of any data type. Therefore, as a last step, scroll through all the columns and decide whether this column is indeed of data type text or if you need to change it accordingly. Here is the corresponding code example:

```
#"Changed Type2" = Table.TransformColumnTypes(#"Pivoted Column",
  {{"revenue", Int64.Type}, {"firstPurchase", type date}, {"ID", Int64.Type}})
```

As a one-time effort, this would be OK. But remember, people decides to create a key-value pair table by hand because schema changes (adding new rows with new keys) are very easy.

When I face such a table, I assume that I have to regularly check the Change Type step and reset the data type for columns where I'd assumed the wrong data type, and set the correct type for new columns as well.

Using M Code

Wouldn't it be better if the key-value pair table contained a hint about the data type of a key? Then you'd be able to use this information to let Power Query set the correct data type automatically. Guess what: that's what the column Type is about!

Many key-value pair tables contain such a column (because the creator of the table also needs a way to track the true data type; e.g., to show a date picker in the UI for a column of data type Date). It's important that you get a list of all possible values for the Type column. You can create a distinct list of the column, but you never can be totally sure that it will be complete. Therefore, you need to talk to the owner of the table for an explanation of what values to expect and how to interpret them. For example, in one of my projects, "1" means "text" and "2" means "decimal number." In the example file, I use values that match the Power Query data types 1:1.

In this improved version, I replace the step Changed Type2 with a step that groups over the Key column and finds the minimal Type. In theory, there'll be only one type per key. Practically, the table contains several rows per key and could therefore contain—by mistake—different types. In such a scenario, I'd pick the alphabetical first type assigned to the key:

```
#"Column Types" = Table.Group(Source, {"Key"}, {{"Type", each List.Min([Type]),
  type nullable text}}),
```

The result of this step is not used in the next step but is referred to several times over the course of the remaining script.

Next, I create blocks of four steps each per data type. There is one block per data type available in Power Query: Text, Int64.Type, Number, Currency, Percentage, Date/

Time, Date, Time, Date/Time/Timezone, Duration, Logical, and Binary. Each of the blocks contains these four steps:

<Datatype> Rows

Selects all rows from step Column Types whose data type should be changed to <Datatype>. The condition puts a filter on column Type according to what the documentation for the key-value pair table tells you. If a type of “1” means it’s a string, then you have to set the condition for step Text Rows to filter where Type equals 1. Remember that Power Query is case sensitive. A check for equality to “text” (with a lowercase t) will not find rows that contains “Text” (with an uppercase T).

<Datatype> Rows Keys

Removes all columns from the previous step, except for the column Key.

<Datatype> Column List

Converts the content of the previous step into a List.

<Datatype> Changed Type

Sets the appropriate data type. A nested Table.TransformColumnTypes and List.Transform allows you to change the data type for a list of column names. This list is not hardcoded but references the list generated in the previous step.

I included the block for integers here:

```
#"Int64.Type Rows" = Table.SelectRows("#Column Types", each ([Type] =
    "Int64.Type")),
#"Int64.Type Rows Keys" = Table.SelectColumns("#Int64.Type Rows",{"Key"}),
#"Int64.Type Column List" = #"Int64.Type Rows Keys"[Key],
#"Int64.Type Changed Type" = Table.TransformColumnTypes ( #"Text Changed Type",
    List.Transform("#Int64.Type Column List", each {_, Int64.Type})),
```

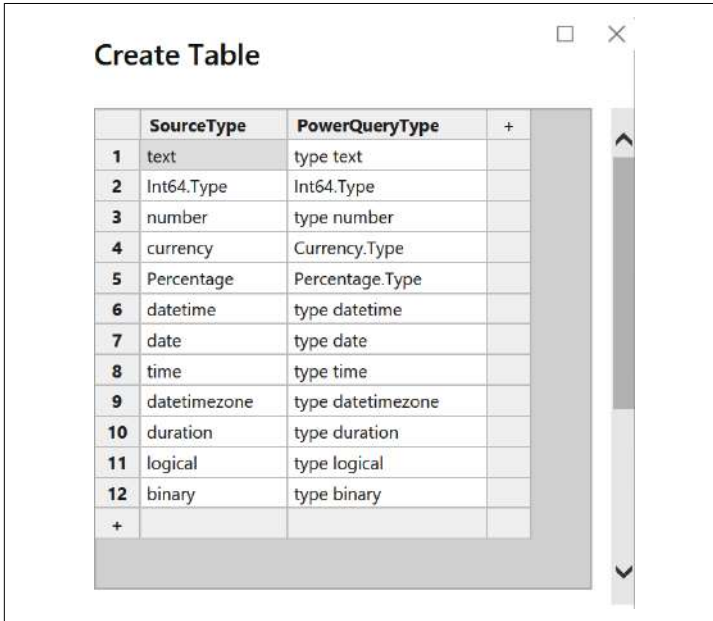
You can take this script and re-use it. The only thing you need to take care of are the <Datatype> Rows steps—remember to update the filter condition according to the content of the Type column.

Writing an M Function

To make everything even easier to use and maintain, I built an even more dynamic solution based on the work of Imke Feldmann and Daniil Maslyuk. First, the matching between the data types mentioned in the key-value pair table and the actual Power Query data types is done via a table. Second, if the type can’t be found in this matching table, the solution finds a fitting data type itself. Third, everything is packaged into a bunch of functions so that you only need to call a function.

Table SourceType2PowerQueryType contains one row per Type delivered from the data source and matches it with the appropriate data type in Power Query. For this

example, use the Enter Data feature to create the pairs (Figure 15-20). In the last step (TypeFromText), call Power Query function TypeFromText to convert the text in column PowerQueryType into a Power Query data type. This will later allow you to convert a Value column into the appropriate data type.



	SourceType	PowerQueryType	+
1	text	type text	
2	Int64.Type	Int64.Type	
3	number	type number	
4	currency	Currency.Type	
5	Percentage	Percentage.Type	
6	datetime	type datetime	
7	date	type date	
8	time	type time	
9	datetimezone	type datetimezone	
10	duration	type duration	
11	logical	type logical	
12	binary	type binary	
+			

Figure 15-20. A table containing the mapping between the source table's Type and the matching Power Query data type

Power Query function TypeFromText takes a text prompt as a parameter and converts it into a Power Query type. It's used in Power Query SourceType2PowerQueryType:

```
let
    /* Based on a script from Imke Feldmann/Daniil Maslyuk
    * https://www.thebiccountant.com/2019/11/17/
    * dynamically-create-types-from-text-with-type-fromtext/#comment-1507
    */
    func = (TypeAsText as text) =>
    Record.Field(
    [
        type null = type null,
        type number = type number,
        Currency.Type = Currency.Type,
        Percentage.Type = Percentage.Type,
        Int64.Type = Int64.Type,
        type datetime = type datetime,
        type date = type date,
        type time = type time,
        type datetimezone = type datetimezone,
```

```

        type duration = type duration,
        type text = type text,
        type logical = type logical,
        type binary = type binary,
        type type = type type,
        type list = type list,
        type record = type record,
        type table = type table,
        type function = type function,
        type anynonnull = type anynonnull
    ],
    TypeAsText)
in
Value.ReplaceType(func, Value.ReplaceMetadata(Value.Type(func),
documentation))

```

Function `TypeFromAny` accepts parameters `TableName` (the name of a Power Query result to process), `Key` for the name of the key column of the mentioned table, `Value` for the value column, and `Type` for the type column. The function first converts all the values explicitly to data type `Any` (Changed to `Any`). Then it removes the `Type` column (`Remove Type`) and pivots the result (`Pivoted Column`), like the other two solutions for the key-value pair problem, presented in the two previous sections.

Next comes the secret sauce of my solution: in step `Added Custom` I try (and catch) to convert the values of a `Key` into different data types. First, try `logical`. If it fails, try `datetime`; if that fails, try `currency`; and as a final fail-safe, assign data type `text`. This solution isn't perfect (e.g., a conversion to date, time, or `datetime` will succeed for any Date/Time-related data and will, therefore, not be distinguished). In this case, choose `datetime` because it can host a date and time as well. A similar thing is true for numbers: any number can be successfully converted into a `decimal`, `currency`, or `whole number`, where I chose `currency` as the common denominator. The code looks like the following:

```

(TableName as table, Key as text, Value as text, Type as text) =>
let
    Source = TableName,
    #"Changed to Any" = Table.TransformColumnTypes(Source,{{Value,
        type any}}),
    #"Remove Type" = Table.RemoveColumns(#"Changed to Any",{Type}),
    #"Pivoted Column" = Table.Pivot(#"Remove Type",
        List.Distinct(#"Removed Columns"[Key]), Key, Value),
    #"Added Custom" = Table.AddColumn(#"Pivoted Column", "Custom", each
        if (try Logical.From(Record.Field(_, Record.Field(_, Key))) catch (r)
            => null) <> null then "logical" else
        if (try DateTime.From(Record.Field(_, Record.Field(_, Key))) catch (r)
            => null) <> null then "datetime" else
        if (try Number.From(Record.Field(_, Record.Field(_, Key))) catch (r)
            => null) <> null then "currency" else
        "text"),

```

```

Custom = #"Added Custom"{0}[Custom]

in
Custom

```

Function `TablePivotDynamic` with parameter `TableName` is very useful; the names for the mandatory columns in this table are `IDColumn`, `KeyColumn`, `ValueColumn`, and `TypeColumn`. This is the function that you should call in your solution because it takes care of all necessary steps to pivot your key-value pair table and gracefully assign the data types to the key columns.

This code is, again, based on the works of Feldmann and Maslyuk. I incorporated the call to function `TypeFromAny` in cases where I couldn't find a match in the table `Source` to `PowerQueryType`. It achieves the work of the blocks-per-data-type in the previous section's solution in a dynamic way:

```

(TableName as table, IDColumn as text, KeyColumn as text,
 ValueColumn as text, TypeColumn as text) =>
let
    Source = TableName,
    #"Removed Type" = Table.RemoveColumns(Source,{TypeColumn}),
    #"Changed to Text" = Table.TransformColumnTypes(#"Removed Type",
        {{IDColumn, Int64.Type}, {KeyColumn, type text}, {ValueColumn,
            type text}}),
    #"Pivoted Column" = Table.Pivot(#"Changed to Text",
        List.Distinct(#"Changed to Text"[Key]), KeyColumn, ValueColumn),

    /*
    inspired by Imke Feldmann
    Dynamic & bulk type transformation in Power Query, Power BI and M
    https://www.thebiccountant.com/2017/01/09/
        dynamic-bulk-type-transformation-in-power-query-power-bi-and-m/
    */

    #"Column Types" = Table.Group(Source, {KeyColumn}, {{TypeColumn,
        each List.Min(Record.Field(_, TypeColumn)), type nullable text}}),
    #"Find missing Type" = Table.AddColumn(#"Column Types", "TypeReplaced",
        each
            if Record.Field(_, TypeColumn) = null or Record.Field(_, TypeColumn)
            = ""
            then TypeFromAny(Source, KeyColumn, ValueColumn, TypeColumn)
            else Record.Field(_, TypeColumn)),
    #"Removed Type 2" = Table.RemoveColumns(#"Find missing Type",{TypeColumn}),
    #"Renamed Type" = Table.RenameColumns(#"Removed Type 2",
        {"TypeReplaced", TypeColumn}),
    #"Changed Type to Text 2" = Table.TransformColumnTypes(#"Renamed Type",
        {{TypeColumn, type text}}),
    #"Lowercased Type" = Table.TransformColumns(#"Changed Type to Text 2",
        {{TypeColumn, Text.Lower, type text}}),
    #"Trimmed Type" = Table.TransformColumns(#"Lowercased Type",
        {{KeyColumn, Text.Trim, type text}}),

```

```

#"Merged Queries" = Table.NestedJoin("#Trimmed Type", {TypeColumn},
    SourceType2PowerQueryType, {"SourceType"}, "SourceType2PowerQueryType",
    JoinKind.LeftOuter),
#"Expanded SourceType2PowerQueryType" =
    Table.ExpandTableColumn("#Merged Queries",
        "SourceType2PowerQueryType", {"PowerQueryType", "DataType"},
        {"PowerQueryType", "DataType"}),
#"List Key & Type" = List.Zip({#"Expanded SourceType2PowerQueryType"
    [Key], #"Expanded SourceType2PowerQueryType"[DataType]}),
#"Set Data Type" = Table.TransformColumnTypes ( #"Pivoted Column",
    #"List Key & Type")
in
#"Set Data Type"

```

The last section of this chapter is dedicated to the role Power Query plays when it comes to comparing self-service and enterprise BI.

Combining Self-Service and Enterprise BI

You would only create transformations in Power Query yourself if you're using Power BI as a self-service BI tool. In an enterprise BI environment, all the necessary transformations are already done by data engineers in the end user's data source: in a data warehouse layer, an Analysis Services database, a Power BI dataflow, an Power BI semantic model, etc. When you discover yourself applying steps in Power BI while you connected to the enterprise data warehouse layer, you should stop and reconsider. Ask yourself why (if?) this transformation is really necessary and why it wasn't already done in the data source. If it is necessary, then talk to the owner of the data source and agree on a solution to add the transformation there. Only add it in Power Query as a "quick fix"—an intermediate solution if the timeline for implementing the transformation in the data source is too long. Set a reminder in your calendar to review the steps in Power Query at a later point to adapt it to use the transformed data from the data source.

I recommend building a centralized enterprise BI solution over self-service BI when information is used more than once in your organization. Every time you transform data, it's only available for others if they connect to this dataset. Not all datasets (and the transformations they contain) can be re-used equally well. Transformation made with Power Query are available in the Power BI semantic model if published to the Power BI service. Users with access to it can connect via Power BI Desktop or Excel. Your mileage with other tools may vary, as other tools (or users) might prefer a relational database.

To satisfy requirements from a broader range of tools (and users), it's a good idea to apply all transformations in a relational data warehouse as a common denominator from which you load the data (without any additional transformations) into a Power BI semantic or Analysis Services tabular model.

Key Takeaways

This chapter presented solutions to practical use cases. It demonstrated that you can take some extra steps in Power Query (mostly by using the power of the M language) to create a dynamic and resilient solution, which doesn't break (so easily) with changes in the data source. Here are some of the key takeaways:

- You can create a bin table as either a distinct list of the facts or by generating a list of values in Power Query. A table containing simply the ranges can be created via the M language's table operator. In both cases, the bin's borders should be provided in Power Query parameters for easy maintenance.
- Just apply the concept of denormalizing to create a bridge table in case a fact table doesn't have the same granularity as the dimension table.
- The Textcomponent needs to be pivoted for easy usage in the report. This can be done as a simple transformation step.
- You can use Azure Cognitive Services to translate the Textcomponent table.
- A key-value pair table needs to be pivoted in most cases so that common reports can be built on top of it. This isn't a real challenge in Power Query and can be solved with the UI. It's very important to set the right data type per key, though. I showed you a script that uses the Type column's information to automatically apply the data type, and another script that can even find a fitting data type per Key automatically.
- Power Query is the go-to tool to apply transformations in a self-service BI scenario. In an enterprise BI setting, consider moving transformations into the data warehouse layer (and just load the tables, then, 1:1 into Power BI without further transformations).

This was the third of four chapters about Power Query. In the next chapter, you will learn how to support the performance tuning concepts discussed in [Chapter 8](#) in Power Query.

Performance Tuning the Data Model with Power Query

In this chapter, you will learn how to improve the performance of a data model with different strategies in Power Query. You will learn that Power Query is not available in all *storage modes*; the choice of storage mode is therefore important. Depending on the storage mode, partitioning can tremendously speed up refresh and/or query time. I will show you how you can support your partitioning strategy in Power Query.

Finally, this chapter talks about another strategy to improve query time: how to *pre-aggregate* the content of a table. Remember, aggregation tables are tables with a different granularity than the base (transaction) table. This can speed up calculations. For example, you can create a table that aggregates calculations by day. Then the value for a year does not need to be calculated based on millions of rows but based on only 365 pre-aggregated values. You can use the *Performance Tuning.pbix* file to follow along with the examples in this chapter.

Storage Mode

The aggregation table implemented with Power Query can be in any of the available storage modes (Import, DirectQuery, or Dual), except for live connection, which prohibits the use of Power Query at all. That means that the aggregation table does not necessarily need be imported into the data model but can be “virtualized” in DirectQuery mode. This is feasible if you need not only to keep the transaction table in DirectQuery mode (see [Chapter 8](#) for a refresh on when using DirectQuery makes sense) but the aggregation table as well.

In a perfect world, Power Query’s feature *query folding* will help to apply aggregations directly to the query sent to the data source so that no explicit aggregation table should be necessary. If query folding doesn’t happen, the whole content of the Source step in Power Query is transferred from the data source, and then all transformations listed in “Applied steps” are executed locally on your machine. This isn’t very efficient, especially when you work with a “smart” data source, like a database system, which could apply transformations on the server’s side. In real-world scenarios, query folding might generate a less-than-optimal query, though. Then, an aggregation table created in Power Query can be of help.



Chapter 20 shows how you can create an aggregation table in a relational database. In combination with DirectQuery, this might be the better solution than Power Query. I would use Power Query in combination with DirectQuery for aggregation tables only if adding a database object to the data source is not an option.

At the time of writing, Power BI’s “Manage aggregation” feature can be applied only to a base table that’s in DirectQuery mode. The aggregation table must be in Import mode. **Chapter 12** shows that you can manage which aggregation to use in DAX too. Therefore, any combination of storage modes (except for live connection) can be made for the base table and “its” aggregation table(s).

Independent from the storage mode, a table contains one or more partitions—subparts which can be managed and refreshed independently from each other.

Partitioning

By default, every refresh in Power BI triggers a full load of the table. To implement a delta load, you need either to partition your table in Power BI (usually by date; see “**Partitioning**” on page 160) and implement a logic to only refresh those partitions where you expect that changes have happened.

Natively, Power BI Desktop doesn’t allow you to specify these partitions. And even with external tools, you can’t specify partitions in your Power BI Desktop file. Instead, you need to publish your *.pbix* file to the Power BI service first. At the time of writing, you need a workspace with premium capabilities in order to create partitions (as read/write access to the XMLA endpoint is needed). Then you need an external tool, like Tabular Editor, to define the partitions.

Custom partitions give you great flexibility in creating the definitions and great control over when to trigger a refresh for a particular partition. Think of a partition as a sub-element of a table. Indeed, when you don’t explicitly partition a table, it consists of one single partition. Every partition has its “own” copy of the Power Query/M script to refresh it, with the exception that at least one filter is different.



Ensure that filters are written (and tested!) in a way that no row of the table gets lost (because then you'd load too few rows from the source) or can be part of more than one partition (because then you'd load this row into multiple partitions of the same table). Both would falsify the table's content and must be avoided in all circumstances. You are responsible, totally on your own, for the partitions and their filters—I'm not aware of any tool that will check the partitions for the mentioned mistakes for you.

Alternatively, you can use a feature of the Power BI service called *incremental refresh*. Every time a refresh is triggered, the Power BI service will take care to refresh (only) the necessary partitions. To use this feature, you need to create two mandatory Power Query parameters of data type Date/Time. The names of these two parameters are RangeStart and RangeEnd.

You need to create filters with the help of these two parameters in all queries that you intend to refresh incrementally. For example, you would add a filter to a query and specify that the OrderDate column must be equal to or after the RangeStart's value and before the RangeEnd's value. After that, you can activate the incremental refresh settings in Power BI's Model view under the table's properties.



No matter how you define partitions in Power BI, make sure to align the partitions with the partition strategy in the data source. If the data source has different or no partitions, the danger is that a refresh for each partition in Power BI will trigger a full scan of the whole table in the data source. This will make the overall refresh of the data model *slower* than no Power BI partitions—the opposite of what you want to achieve with partitioning.

Partitioning makes the refresh faster—when the filters of a query match the partition key, it can also make a query faster. However, a query becomes even faster when it is built on top of pre-aggregated data.

Pre-Aggregating

Pre-aggregating means that you create a Power Query that applies grouping on some of the columns of the Power Query and applies aggregation functions on others. You can achieve this via Home → Group By in the Power Query window. The Basic mode of the dialog window (Figure 16-1) allows you to specify one column to group on and have one column created based on an aggregation Operation. You can give this column a name of your choice (“New column name”).

Group By

Specify the column to group by and the desired output.

☒ Basic ☐ Advanced

OrderDate

New column name: Sales Amount

Operation: Sum

Column: SalesAmount

OK Cancel

Figure 16-1. Grouping by OrderDate and aggregating the Sales Amount

You can apply any of the following operations:

- Sum
- Average
- Median
- Min
- Max
- Count Rows
- Count Distinct Rows
- All Rows

For the majority of operators, you need to specify a column on which the aggregations will be applied (e.g., the content of which column you want to sum up). The latter three operators don't allow you to specify a column, but they are calculated over all columns, counting all the rows (Count Rows), counting all the rows with a unique combination of values (Count Distinct Rows), or collapsing everything into a column of type Table (All Rows).

In Advanced mode, you can add several columns on which you want the resulting query to be grouped by, and add more aggregations. In [Figure 16-2](#), I keep the aggregation to a single column (OrderDate) but add two aggregations: one Sum of the SalesAmount, which I name SalesAmount, and another named SalesCount, which just counts the rows per OrderDate.

The result is a query with three rows: OrderDate, SalesAmount, and SalesCount, which has one single row per OrderDate. If there is no filter applied, or only Order Date is filtered, visuals based on this aggregated query will be way faster compared to applying the same calculations on the Reseller Sales table.

Group By

×

Specify the columns to group by and one or more outputs.

☐ Basic
 ☒ Advanced

OrderDate

▼

Add grouping

New column name

Operation

Column

SalesAmount

Sum

SalesAmount

SalesCount

Count Rows

Add aggregation

OK

Cancel

Figure 16-2. Grouping by *OrderDate*, aggregating the *SalesAmount*, and calculating the number of rows



“Pre-Aggregating” on page 166 describes how you can make Power BI aware of the available aggregations. If the logic is more complex, see “Pre-Aggregating” on page 235 for more on how to change your measures in order to calculate in the most optimal way.

Key Takeaways

This chapter showed steps in Power Query that can support the performance tuning strategy for your data model. Here are the key takeaways:

- Tables created in Power Query can be in either import, DirectQuery, or Dual storage mode. Therefore, an aggregation table created with Power Query can be imported (for best performance) or stay in DirectQuery mode (or Dual for that matter) if you have reasons to do so. Live connections do not allow for any transformations in Power Query.
- Power BI’s incremental refresh builds on top of two Power Query parameters and filters you apply to a Power Query. Partitions are then automatically created for you.

- You can create your own custom partitions as well. You are fully responsible for setting the filters so that every row appears in exactly one partition.
- Partitions in Power BI should always be aligned with the partitions in the data source to gain performance advantages (otherwise the performance can degrade instead of improving).
- Power Query offers a limited list of aggregation functions you can apply on existing columns: sum, average, median, min, max. You can also count all or the distinct rows or collapse everything into a column of type Table.

This chapter closes this book's coverage of Power Query. **Part V** is dedicated to the SQL language. You will learn what you can do in a relational data warehouse (layer) in terms of supporting the perfect data model for Power BI.

Data Modeling for Power BI with the Help of SQL

Chapters	Parts				
	Part 1 Data modeling in general	Part 2 Power BI	Part 3 DAX	Part 4 Power Query	Part 5 SQL
Understanding a data model	Chapter 1	Chapter 5	Chapter 9	Chapter 13	Chapter 17
Building a data model	Chapter 2	Chapter 6	Chapter 10	Chapter 14	Chapter 18
Real-world examples	Chapter 3	Chapter 7	Chapter 11	Chapter 15	Chapter 19
Performance tuning	Chapter 4	Chapter 8	Chapter 12	Chapter 16	Chapter 20

The part about SQL is especially aimed toward data engineers (usually part of the IT department) and the dedicated domain expert. SQL stands for “Standard Query Language,” a deceiving name, as every database management system comes with its own dialect. As this book is about Microsoft Power BI, I will concentrate on Transact-SQL (T-SQL), which is available for all SQL interfaces in Microsoft’s data platform. This dialect comes with a procedural extension, which allows to create variables,

implement conditional executions of code or loops. Such a code can be stored in the form of procedures and functions in a database.

Chapter 17 starts with an introduction to the parts of a data model in a SQL-based database:

- Tables
- Primary and Foreign Keys
- Relationships
- Combining the content of tables and possible traps

Chapter 18 shows you all steps typically used when building a data warehouse (layer) from any data source:

- Normalizing and Denormalizing
- Adding calculations
- Transforming flags and indicators into meaningful text
- Creating your own date and time tables
- Duplicating tables in case they play more than one role in a data model
- Implementing *slowly changing dimensions* of different types
- Flattening parent-child hierarchies

The challenges of the real-world can be manifold. Also in this part, I will show you how to solve the four introduced in **Chapter 3** by demonstrating the power of the SQL language:

- Binning
- Multi-fact data models
- Multi-language data models
- Key-Value pair tables

This part concludes with solutions in SQL to support a good performing data model in Power BI (**Chapter 20**). On the one hand, you can decide to “virtualize” transformations in SQL or persist the data in the right shape in the relational database. On the other hand, you need to decide if you can import the data provided by SQL into Power BI or only query it when needed in a Power BI visual.

Understanding a Relational Data Model

Relational databases have existed since 1970 and have introduced a lot of new concepts: tables, relationships, constraints, normalization, etc. The concept, and its implementation by various vendors (e.g., SQL Server, Azure SQL DB, or Azure SQL Managed Instance by Microsoft), is still successful and allows for a variety of use cases. That's why you find both application databases (OLTP) and analytical databases (OLAP) implemented as relational databases.

This chapter guides you on how relational databases are different from Power BI and Analysis Services tabular. Due to these differences, you will learn that a relational data warehouse is the perfect addition to your analytical infrastructure in an enterprise environment. I explore techniques, use cases, and how to implement them in a relational database (managed and updated by SQL) to make the experience in Power BI and Analysis Services tabular great.

I introduce some basic concepts: that a data model consists of tables, that columns of a table can have different purposes (key or attribute), and how you can combine information that is spread out into different tables.

Data Model

A data model implemented in Power BI/Analysis Services tabular and one implemented in a relational database have many things in common. You store both data and metadata in it. The data is hosted in tables. Metadata explains how the tables form the data model. Let's introduce the basic parts of a relational data model.

Basic Components

This section introduces you to the basic components of a relational data model:

- Tables
- Relationships
- Primary keys
- Surrogate keys
- Foreign keys

Tables

Tables are the core of a relational database. They're where the data is stored. A table is part of a database schema, which is part of a database, which is stored on a database server. The database schema works well as a security barrier. I put all the content I want to expose to Power BI into its own schema and give the users and applications read-only access to (only) this schema. Typically, I name this schema *PowerBI*, *Reporting*, or something similar. `[PowerBI].[Sales]` would refer the *Sales* table within the *PowerBI* schema.

You have influence over the physical storage of a table by creating a clustered index on it. As a rule of thumb, all fact tables and dimension tables with a size over one million rows should be stored in the columnstore format. All other tables should have clustered indexes on their primary keys and page compression enabled.

To further improve performance for different scenarios, you might create non-clustered index. As long as you plan to fully load the data from the data warehouse into the Power BI/Analysis Services data model (which is the recommendation, unless you have to specific reasons to not do so), and nobody and no application is directly querying the tables, there is no need to create such additional index.

A table consists of columns. The columns can store data of different types. When working in SQL Server/Azure SQL DB, data can be organized in the following ways:

Exact numerics

Exact numerics include `bit`, `tinyint`, `smallint`, `int`, `bigint`, `numeric`, `decimal`, `smallmoney`, and `money` types. They can be stored in Power BI's `Whole number`, `Fixed decimal number`, or `Decimal number`. A value of SQL's type `bit` is usually stored as `True/False` in Power BI. `smallmoney` and `money` are legacy data types and should not be used anymore—choose one of the other data types instead. `numeric` and `decimal` are synonyms for each other. You need to provide scale and precision, e.g., `decimal (5,2)` can fit values between `-999.99` and `+999.99`—a precision of up to five digits, of which are two decimals. The default precision is 18; the default scale is 0.

Approximate numerics

Approximate numerics include `float` and `real` and can be stored in Power BI's `Decimal number`. You should avoid these approximate data types altogether, as they are not exact, and unexpected rounding effects can bite; use one of the exact numerics instead.

Date and time

Date and time data types include `datetime`, `smalldatetime`, `datetime2`, `datetimeoffset`, `date`, and `time`. These can be stored in Power BI's `Date/Time`, `Date`, or `Time` data type. `datetime` and `smalldatetime` are legacy data types you should not use anymore—use one of the other three data types instead. `datetime2`, `datetimeoffset`, and `time` can have a precision of up to 7 fractions of a second (100 nanoseconds). This is also the default if you are not providing a precision.

Character strings

Character strings include `char`, `varchar`, `text`, `nchar`, `nvarchar`, and `ntext` and can be stored in Power BI's `Text` data type. `text` and `ntext` are legacy data types and should not be used anymore; they should be replaced by `varchar(max)` or `nvarchar(max)`, respectively.

For all text columns, you need to specify a size (otherwise it defaults to 1). The maximum is 8,000 bytes. When the string size might exceed 8,000 bytes, use “max” as the precision. The character string data types with a preceding `n` (`nchar` and `nvarchar`) are stored in the Unicode format. One character occupies two bytes; you can only fit half of the characters per byte compared to the ordinary character string data types.

Binary strings

Binary strings include `binary`, `image`, and `varbinary`. They can be stored in Power BI's `Binary` data type, but it's better to remove a column of this data type before loading into Power BI, as it's not supported.

Other data types

Other data types include `cursor`, `hierarchyid`, `sql_variant`, `spatial geometry`, and `geography` types, `table`, `rowversion`, `uniqueidentifier`, and `xml`. These are not supported in Power BI. You need to extract the necessary information into one of the supported data types, before importing into Power BI (e.g., the relevant parts of the `xml`, or resolving the hierarchical information from a `hierarchyid` column).

The best practice is to not directly expose the tables of a data warehouse but its views instead. Simply put, views are stored `SELECT` statements, which expose the content of one or more tables. From the outside perspective, queries against tables and queries

against views can't be distinguished from each other. The only disadvantage is that you can't create foreign key constraints on views. Therefore, Power BI (or Analysis Services tabular) can't create filter relationships based on this information (but needs to apply other rules to discover them for you, if you want to do so). The advantage is that the views work as an additional layer between the physical relational data model and Power BI. If there are changes to the physical relational data model, the views can be changed accordingly so that they still return the same content. This eases the roll-out of changes because Power BI (or any other tool that uses the relational database as a data source) doesn't have to change how it accesses the information.

I consider views to be equivalent to an API. If I really need to implement (breaking) changes, I create a new database schema and create the new set of views there. This gives all report creators a grace period to migrate their reports to the new schema/data model before I turn off the "old" schema.

Alternatively, you could also expose data in the form of a stored procedure or a table-valued function. Stored procedures are called via the EXEC keyword. Table-valued functions are called as part of a query (either in the FROM or the JOIN part). Both can be useful in edge cases, as you can provide parameters to a stored procedure or a function, making for perhaps more efficient code inside the procedure or function. I prefer not to use them because they mean writing (a small piece) of SQL as the data source in Power BI, which needs to be maintained or may include some logic. I think I'm better off with "ordinary" views and tables to which I can simply connect in Power BI/Power Query.

Relationships

Queries in SQL do not depend on any up-front definition of a relationship. You must define the necessary relationship via the JOIN operator in each and every query. This gives you the freedom to apply any JOIN as you need it in the specific query (including non-equi-joins).

You can define foreign key constraints (we'll talk about them later in this chapter). But they give only suggestions to the query author and do not limit the kind of queries you can write. With foreign key constraints given, you must still define the (kind of) joins in each and every query.

Primary Keys

Primary keys can be explicitly defined as a constraint on a table (PRIMARY KEY). This constraint is a combination of a UNIQUE constraint and a CHECK constraint under the hood of the database system. The UNIQUE constraint will limit the content of the column to only unique values (prohibiting duplicates from being created via INSERT or UPDATE). And a check constraint to disallow NULL values for the column is created on

top (NOT NULL). This is strongly recommended, so that you can discover data quality issues right away, as soon as the table is manipulated.

Here's the definition of the product's table primary key:

```
ALTER TABLE [dbo].[DimProduct]
ADD CONSTRAINT [PK_DimProduct_ProductKey] PRIMARY KEY CLUSTERED
(
    [ProductKey] ASC
) ON [PRIMARY]
```

In a data warehouse, the primary key should be a surrogate key (read on to learn how you create a surrogate key in a SQL database). Additionally, you should create a unique constraint/unique index on the business key of a table. This will speed up queries that filter on the business key (as it will be necessary to look up the primary key for a business key in a dimension table during the ETL process).

Surrogate Keys

In “[Surrogate Keys](#)” on page 8, I emphasize that having a single integer surrogate key as the primary key of a table is best practice. In Azure SQL, you can define a surrogate key quite easily by using the keyword `IDENTITY(1, 1)` for the primary key column when you create a table. This lets the database automatically maintain unique numbers for your primary key. The first parameter in the keyword means that the first row will get “1” as the content, and the second parameter means all the subsequent rows will get a number that is 1 higher than the previous one. (To be honest, I don't see any reason to start at a number other than 1 or to let the database intentionally generate key numbers with gaps; so “1,1” is simply best practice.)

Due to performance optimization, there could be gaps in the numbers (e.g., that after number 2 the next number is not 3, but 4 or 5). But this isn't that big of an issue because the surrogate key is meaningless by itself and should therefore not be used in any filter or grouping. A signed integer can hold values up to 2 billion—this is usually sufficient for most of the dimension tables, even if there are gaps in the surrogate key. If you can foresee that a table will contain more than 2 billion rows, you should create the primary key as a `BIGINT`, which holds numbers up to 263 (which is over 9 quintillion, a number with 19 digits).

The following example shows how to create a Product table in schema form in PowerBI with a `PRIMARY KEY` called ProductID of type `INT` with the aforementioned `IDENTITY` definition:

```
CREATE TABLE PowerBI.Product (
    ProductID INT IDENTITY(1, 1) PRIMARY KEY,
    Product NVARCHAR(50),
    Subcategory NVARCHAR(50),
    Category NVARCHAR(50)
)
```

Foreign Keys

A foreign key references the primary key of another table—the foreign key is a primary key of a different table. If you create a foreign key constraint, it will guarantee that the foreign key will always match a row in the table containing the referenced primary key. You won't be able to change the foreign key column to a value that can't be found as the primary key in the referenced table. And it will prohibit you from deleting a row in the table containing the primary, which is currently referenced as a foreign key. This is strongly recommended to discover data quality issues as soon as the tables are manipulated.

Creating foreign key constraints is the only way to fully guarantee that there is referential integrity between the tables. Together with disabled nullability, it will allow for inner joins (instead of outer joins), which are faster.

You can add a foreign key constraint to an existing table—DimProduct in this example. The constraint's name (FK_DimProduct_DimProductSubcategory) is mentioned directly after keywords **ADD CONSTRAINT**. After the keywords **FOREIGN KEY**, you need to mention the referencing column in parentheses (ProductSubcategoryKey). The referenced table and column in parentheses ([dbo].[DimProductSubcategory] ([ProductSubcategoryKey])) follow the **REFERENCES** keyword:

```
ALTER TABLE [dbo].[DimProduct] WITH CHECK
ADD CONSTRAINT [FK_DimProduct_DimProductSubcategory]
FOREIGN KEY([ProductSubcategoryKey])
REFERENCES [dbo].[DimProductSubcategory] ([ProductSubcategoryKey])
GO
```

Alternatively, you can create the foreign key constraint during the **CREATE TABLE** statement, as shown in the following code:

```
CREATE TABLE [dbo].[DimProduct](
    [ProductKey] [int] IDENTITY(1,1) NOT NULL,
    [ProductAlternateKey] [nvarchar](25) NULL,
    [ProductSubcategoryKey] [int] NULL REFERENCES [dbo].
    [DimProductSubcategory] ([ProductSubcategoryKey])
,
    ...
);
```



It's best practice to use the prefix FK for foreign keys and to give them a meaningful name. If a constraint is violated during an insert, update, or delete, the name of the constraint is shown. If the name already gives you a hint as to what the constraint is about, it's easier to find out what violated the constraint.

I strongly recommend creating the foreign key constraints in the development environment to quickly discover data quality issues in the ETL process that would violate the constraint. If I need to performance tune the database to the last bits, I keep those constraints, but disable them in the production environment. This allows tools and people to recognize the relationships, without slowing down data manipulation language operations (insert, update, and delete); they'll ignore a disabled constraint. Here is the code to disable and enable a foreign key constraint:

```
ALTER TABLE [dbo].[DimProduct]
NOCHECK CONSTRAINT [FK_DimProduct_DimProductSubcategory];
```

Now that you've learned how to create tables and their relationships in the table's definition, let's talk about combining data that resides in different tables.

Combining Queries

No data model consists of just a single table but always comprises several tables. To extract the necessary information from a data model, you need to combine tables in your queries. This can be either done with set operators or with joins. This section uses the *103 SET and JOINS.sql* file for examples.

Set Operators

SQL allows for all three set operators, which “Set Operators” on page 11 discusses. This section gives examples of all the different set operators (see Tables 17-1 and 17-2). All of them build upon the following two queries:

```
SELECT SalesTerritoryRegion FROM dbo.DimSalesTerritory
```

Table 17-1. The 11 rows of the DimSalesTerritory table

SalesTerritoryRegion
Northwest
Northeast
Central
Southwest
Southeast
Canada
France
Germany
Australia
United Kingdom
NA

```
SELECT DISTINCT EnglishCountryRegionName FROM dbo.DimGeography
```

Table 17-2. The 6 rows of the *DimGeography* table.

EnglishCountryRegionName
Australia
Canada
France
Germany
United Kingdom
<i>United States</i>

For the following subsections, understand that the two queries' results have both identical rows (Australia, Canada, France, Germany, and United Kingdom), and rows that appear only in one of the results. The first query contains Northwest, Northeast, Central, Southwest, Southeast, and NA exclusively. The second query contains United States exclusively. I put the rows that are specific for one table in *italic*. I will describe the result of the combined queries for each set operator.

UNION

The UNION operator puts the two queries just underneath each other to form one list of items. If you want to keep duplicates (or if you are sure that there can't be duplicates under any circumstances), you can save the database from an (expensive) sort operation, which would be needed to look for duplicates were you using UNION ALL instead of UNION. The first query (UNION) only returns 12 rows (see Table 17-3):

```
-- UNION
SELECT SalesTerritoryRegion FROM dbo.DimSalesTerritory
UNION
SELECT DISTINCT EnglishCountryRegionName FROM dbo.DimGeography
```

Table 17-3. The resulting 12 rows of the “UNIONed” queries

SalesTerritoryRegion
Australia
Canada
Central
France
Germany
NA
Northeast
Northwest
Southeast

SalesTerritoryRegion

Southwest
United Kingdom
United States

The five duplicate regions (Australia, Canada, France, Germany, and United Kingdom) only appear once in the result. The second query (`UNION ALL`) returns 17 rows (11 rows from the first query plus 6 rows from the second, showing five rows twice; see [Table 17-4](#)):

```
-- UNION ALL
SELECT SalesTerritoryRegion FROM dbo.DimSalesTerritory
UNION ALL
SELECT DISTINCT EnglishCountryRegionName FROM dbo.DimGeography
```

Table 17-4. The 17 rows resulting from the “`UNION ALL`ed” queries

SalesTerritoryRegion

Northwest
Northeast
Central
Southwest
Southeast
Canada
France
Germany
Australia
United Kingdom
NA
Australia
Canada
France
Germany
United Kingdom
United States

INTERSECT

This operator looks for rows that appear in both queries and filters the other rows out. The rows only appearing in the first (Northwest, Northeast, Central, Southwest, Southeast, and NA) or in the second query (United States) are not shown in the following example (see also [Table 17-5](#)):

```
-- INTERSECT
SELECT SalesTerritoryRegion FROM dbo.DimSalesTerritory
INTERSECT
SELECT DISTINCT EnglishCountryRegionName FROM dbo.DimGeography
```

Table 17-5. Five rows as the result of the “INTERSECTed” queries

SalesTerritoryRegion
Australia
Canada
France
Germany
United Kingdom

EXCEPT

This operator looks for rows that appear in either or both queries. But it filters these rows out and only returns rows from the first query result that do not appear in the result of the second query. Only the “extra” rows from the first query are shown (Northwest, Northeast, Central, Southwest, Southeast, and NA). The “extra” rows from the second query are not returned (see also [Table 17-6](#)):

```
-- EXCEPT
SELECT SalesTerritoryRegion FROM dbo.DimSalesTerritory
EXCEPT
SELECT DISTINCT EnglishCountryRegionName FROM dbo.DimGeography
```

Table 17-6. Five rows as the result of the “EXCEPTed” queries

SalesTerritoryRegion
Central
NA
Northeast
Northwest
Southeast
Southwest

Joins

All joins discussed in “[Joins](#)” on [page 13](#) can be implemented in SQL. Again, I built all examples on the same two tables I used to demonstrate the set operators.

INNER JOIN

For an inner join, you need to specify a *join predicate* (a Boolean condition to specify if two rows are related). In the following example (see also [Table 17-7](#)), only rows

with regions (EnglishCountryRegionName and SalesTerritoryRegion) available in both queries are shown in an INNER JOIN (Australia, Canada, France, Germany, United Kingdom, and United States). The INNER keyword is optional and does not change the behavior of the join operation. Whether you specify INNER JOIN or just JOIN, you will get the exact same result:

```
-- INNER JOIN
SELECT DISTINCT dst.SalesTerritoryRegion, dg.EnglishCountryRegionName
FROM dbo.DimSalesTerritory dst
INNER JOIN dbo.DimGeography dg ON
    dg.EnglishCountryRegionName = dst.SalesTerritoryRegion
```

Table 17-7. Five rows as the result of the “INNER JOINed” queries

SalesTerritoryRegion	EnglishCountryRegionName
Australia	Australia
Canada	Canada
France	France
Germany	Germany
United Kingdom	United Kingdom

OUTER JOIN

This operator is available as LEFT, RIGHT, and FULL OUTER JOIN. The OUTER keyword is optional and does not change the behavior of the join operation. Whether you specify LEFT OUTER JOIN or only LEFT JOIN doesn’t matter; it will work and return the exact same result in both cases.

The LEFT OUTER JOIN returns all rows of the first query (see also Table 17-8). For regions not available in the second query, the EnglishCountryRegionName is NULL:

```
-- LEFT OUTER JOIN
SELECT DISTINCT dst.SalesTerritoryRegion, dg.EnglishCountryRegionName
FROM dbo.DimSalesTerritory dst
LEFT OUTER JOIN dbo.DimGeography dg ON dg.EnglishCountryRegionName =
    dst.SalesTerritoryRegion
```

Table 17-8. The 11 rows that result from the “LEFT OUTER JOINed” queries

SalesTerritoryRegion	EnglishCountryRegionName
Australia	Australia
Canada	Canada
Central	NULL
France	France
Germany	Germany
NA	NULL

SalesTerritoryRegion	EnglishCountryRegionName
Northeast	NULL
Northwest	NULL
Southeast	NULL
Southwest	NULL
United Kingdom	United Kingdom

The `RIGHT OUTER JOIN` returns all rows of the second query. For regions not available in the first query, the `SalesTerritoryRegion` is `NULL` (see also [Table 17-9](#)):

```
-- RIGHT OUTER JOIN
SELECT DISTINCT dst.SalesTerritoryRegion, dg.EnglishCountryRegionName
FROM dbo.DimSalesTerritory dst
RIGHT OUTER JOIN dbo.DimGeography dg ON dg.EnglishCountryRegionName =
    dst.SalesTerritoryRegion
```

Table 17-9. Six rows as the result of the “`RIGHT OUTER JOINed`” queries

SalesTerritoryRegion	EnglishCountryRegionName
NULL	United States
Australia	Australia
Canada	Canada
France	France
Germany	Germany
United Kingdom	United Kingdom

The `FULL OUTER JOIN` returns all rows of both queries, no matter whether or not there is a matching row in the other table. For regions not available in the other query, you will get a `NULL` result (see also [Table 17-10](#)):

```
-- FULL OUTER JOIN
SELECT DISTINCT dst.SalesTerritoryRegion, dg.EnglishCountryRegionName
FROM dbo.DimSalesTerritory dst
FULL OUTER JOIN dbo.DimGeography dg ON dg.EnglishCountryRegionName =
    dst.SalesTerritoryRegion
```

Table 17-10. The 12 rows that result from the “`FULL OUTER JOINed`” queries

SalesTerritoryRegion	EnglishCountryRegionName
NULL	United States
Australia	Australia
Canada	Canada
Central	NULL
France	France

SalesTerritoryRegion	EnglishCountryRegionName
Germany	Germany
NA	NULL
Northeast	NULL
Northwest	NULL
Southeast	NULL
Southwest	NULL
United Kingdom	United Kingdom

Anti-join

There is no keyword for an anti-join in SQL. You must implement it by combining an OUTER JOIN with a WHERE clause, which filters only those rows where the join key on the outer rows are NULL. As this is an outer join, you can write it as left, RIGHT or FULL ANTI-JOIN. The LEFT ANTI-JOIN returns rows only appearing in the first query (Northwest, Northeast, Central, Southwest, Southeast, and NA), which means that the EnglishCountryRegionName is NULL (see also [Table 17-11](#)):

```
-- LEFT ANTI-JOIN
SELECT DISTINCT dst.SalesTerritoryRegion, dg.EnglishCountryRegionName
FROM dbo.DimSalesTerritory dst
LEFT JOIN dbo.DimGeography dg ON dg.EnglishCountryRegionName =
    dst.SalesTerritoryRegion
WHERE dg.EnglishCountryRegionName IS NULL
```

Table 17-11. Six rows as the result of the LEFT ANTI-JOINED queries

SalesTerritoryRegion	EnglishCountryRegionName
Central	NULL
NA	NULL
Northeast	NULL
Northwest	NULL
Southeast	NULL
Southwest	NULL

The right anti-join returns rows only appearing in the second query (United States), showing NULL for the SalesTerritoryRegion (see also [Table 17-12](#)):

```
-- RIGHT ANTI-JOIN
SELECT DISTINCT dst.SalesTerritoryRegion, dg.EnglishCountryRegionName
FROM dbo.DimSalesTerritory dst
RIGHT JOIN dbo.DimGeography dg ON dg.EnglishCountryRegionName =
    dst.SalesTerritoryRegion
WHERE dst.SalesTerritoryRegion IS NULL
```

Table 17-12. One row as the result of the *RIGHT ANTI-JOINED* queries

SalesTerritoryRegion	EnglishCountryRegionName
NULL	United States

The *FULL ANTI-JOIN* returns rows only appearing exclusively in either the first or second query (Australia, Canada, France, Germany, and United Kingdom versus United States). Every row shows *NULL* in either *SalesTerritoryRegion* or *EnglishCountryRegionName* (see also [Table 17-13](#)):

```
-- FULL ANTI-JOIN
SELECT DISTINCT dst.SalesTerritoryRegion, dg.EnglishCountryRegionName
FROM dbo.DimSalesTerritory dst
FULL JOIN dbo.DimGeography dg ON dg.EnglishCountryRegionName =
    dst.SalesTerritoryRegion
WHERE dst.SalesTerritoryRegion IS NULL or dg.EnglishCountryRegionName IS NULL
```

Table 17-13. Seven rows as the result of the *FULL ANTI-JOINED* queries.

SalesTerritoryRegion	EnglishCountryRegionName
NULL	United States
Central	NULL
NA	NULL
Northeast	NULL
Northwest	NULL
Southeast	NULL
Southwest	NULL

CROSS JOIN

A *CROSS JOIN* returns the so-called Cartesian product, a combination of all rows of the first query with all rows from the second query (see [Table 17-14](#)). This long list has only certain, narrow use cases.

```
-- CROSS JOIN
SELECT DISTINCT dst.SalesTerritoryRegion, dg.EnglishCountryRegionName
FROM dbo.DimSalesTerritory dst
CROSS JOIN dbo.DimGeography dg
```

Table 17-14. Some of the 66 rows as the result of the *CROSS JOINED* queries

SalesTerritoryRegion	EnglishCountryRegionName
Australia	Australia
Australia	Canada
Australia	France
Australia	Germany

SalesTerritoryRegion	EnglishCountryRegionName
Australia	United Kingdom
Australia	United States
...	...
United Kingdom	Australia
United Kingdom	Canada
United Kingdom	France
United Kingdom	Germany
United Kingdom	United Kingdom
United Kingdom	United States

SELF JOIN

The following example isn't based on the two region tables, but on the employee table. It contains a primary key (EmployeeKey) and a foreign key (ParentEmployeeKey), which refers the primary key of the same table (EmployeeKey). As discussed in [“Joins” on page 13](#), this allows for implementing a hierarchical structure (like an organigram). The same table (DimEmployee) appears twice in a query to implement a self join:

```
-- SELF JOIN
SELECT employee.EmployeeKey, employee.ParentEmployeeKey, employee.FirstName,
       parent.EmployeeKey, parent.FirstName
FROM   dbo.DimEmployee employee
JOIN   dbo.DimEmployee parent ON parent.EmployeeKey = employee.ParentEmployeeKey
```

Table 17-15. Examples from the result of the SELF JOINed queries

EmployeeKey	ParentEmployeeKey	FirstName	EmployeeKey	FirstName
4	3	Rob	3	Roberto
5	3	Rob	3	Roberto
11	3	Gail	3	Roberto
13	3	Jossef	3	Roberto
162	3	Dylan	3	Roberto
267	3	Ovidiu	3	Roberto
271	3	Michael	3	Roberto
274	3	Sharon	3	Roberto
...
295	290	Rachel	290	Amy
291	290	Jae	290	Amy
292	290	Ranjit	290	Amy
296	294	Lynn	294	Syed

EQUI-JOIN

Commonly, you will write most of the joins as equi-joins: you want to find rows where the primary key of one table matches the foreign key of another table. All of the previous examples demonstrating the different kinds of the JOIN keyword were equi-joins, as the condition after the ON keyword contained solely the equal-sign (“=”). That’s why I don’t list an example here.

NON-EQUI-JOIN

Only in rare use cases will you write non-equi-joins: when finding rows matching a range of values, for example. The following example only shows combinations of rows where the region names are not the same. Basically, it looks like the full join, but with the matching rows (e.g., were both SalesTerritoryRegion and EnglishCountry RegionName have the same content) not listed. This is more of an educational example (see also [Table 17-16](#)). Later, when we talk about binning (in [“Binning” on page 399](#)), you will see a more relevant example of an non-equi-join:

```
SELECT DISTINCT dst.SalesTerritoryRegion, dg.EnglishCountryRegionName
FROM   dbo.DimSalesTerritory dst
JOIN   dbo.DimGeography dg ON
       dg.EnglishCountryRegionName <> dst.SalesTerritoryRegion
```

Table 17-16. Some of the 61 rows as the result of the “non-equi-join’ed” queries

SalesTerritoryRegion	EnglishCountryRegionName
Australia	Canada
Australia	France
Australia	Germany
Australia	United Kingdom
Australia	United States
...	...
United Kingdom	Australia
United Kingdom	Canada
United Kingdom	France
United Kingdom	Germany
United Kingdom	United States

Natural joins

This type of join is not supported in Microsoft’s SQL dialect. That means that you must always specify a join predicate (the columns you want to be matched during the join).

Join Path Problems

When you join a row of one table to a row of another table, you can face several problems, resulting in unwanted query results.

Loop

Between the table FactResellerSales and the table DimDate exists more than one foreign key constraint. Some users (and unfortunately many reporting tools) will, therefore, create a combined join predicate, asking for only rows where both the FactResellerSales' OrderDateKey and the ShipDateKey are equal to the DimDate's DateKey. This leads to an empty query, as in the example (see also Table 17-17) there was no sale that was ordered and shipped on the exact same day:

```
SELECT
    SUM(SalesAmount) SalesAmount
FROM
    dbo.FactResellerSales frs
JOIN dbo.DimDate dd ON dd.DateKey = frs.OrderDateKey
AND dd.DateKey = frs.ShipDateKey
```

Table 17-17. This simple loop leads to an empty query result

SalesAmount
NULL

In SQL, you must explicitly solve this problem by rewriting this query. You need to join the DimDate table twice. Once with a join predicated on the FactResellerSales' OrderDateKey, and once more with a join predicated on the the FactResellerSales' ShipDateKey. In this case, specifying an alias (like od or sd, as in the example) is mandatory, to distinguish the two references to the same table (DimDate). This is also called a role-playing dimension, as DimDate plays the role of the order date dimension in one join, and the role of the ship date dimension in the other (see also Table 17-18):

```
SELECT
    SUM(SalesAmount) SalesAmount
FROM
    dbo.FactResellerSales frs
JOIN dbo.DimDate od ON od.DateKey = frs.OrderDateKey
JOIN dbo.DimDate sd ON sd.DateKey = frs.ShipDateKey
```

Table 17-18. Joining twice gets you out of the loop dilemma

SalesAmount
80,450,596.9823

Chasm trap

A join between FactResellerSales and DimDate leads to duplicate rows of the DimDate table, as there is more than one row in FactResellerSales per row in DimDate (remember that the relationship between DimDate and FactResellerSales has a one-to-many cardinality). This is expected and intended. The exact same thing happens if you join FactInternetSales with DimDate.

But when you join all three tables (FactResellerSales, DimDate, and FactInternetSales) in the same query, you will get the rows of FactInternetSales duplicated (per duplicate row in DimDate). But this will also happen in the other direction, as the relationship between DimDate and FactInternetSales is, again, a one-to-many relationship. The result is a very inflated sum of SalesAmount (which results from the true sales amount being multiplied by the number of rows in the “other” table). Let’s first query the two fact tables separately (see also Tables 17-19 and 17-20):

```
SELECT
    dd.DateKey, SUM(frs.SalesAmount) ResellerSalesAmount,
    COUNT(*) ResellerSalesCount
FROM
    dbo.FactResellerSales frs
JOIN dbo.DimDate dd ON dd.DateKey = frs.OrderDateKey
WHERE dd.DateKey=20110101
GROUP BY dd.DateKey
```

Table 17-19. For January 1 2011, there are 786 reseller sales worth 1,538,508.3122

DateKey	ResellerSalesAmount	ResellerSalesCount
20110101	1,538,408.3122	785

```
SELECT
    dd.DateKey, SUM(fis.SalesAmount) InternetSalesAmount,
    COUNT(*) InternetSalesCount
FROM
    dbo.FactInternetSales fis
JOIN dbo.DimDate dd ON dd.DateKey = fis.OrderDateKey
WHERE dd.DateKey=20110101
GROUP BY dd.DateKey
```

Table 17-20. For January 1 2011, there are 2 internet sales worth 7,156.54

DateKey	ResellerSalesAmount	ResellerSalesCount
20110101	7,156.54	2

So far so good. However, if you simply combine the two queries and join all three tables in one query, the results are inflated (see Table 17-21):

```
SELECT
    dd.DateKey, SUM(frs.SalesAmount) ResellerSalesAmount,
```

```

SUM(fis.SalesAmount) InternetSalesAmount
FROM
    dbo.FactResellerSales frs
JOIN dbo.DimDate dd ON dd.DateKey = frs.OrderDateKey
JOIN dbo.FactInternetSales fis ON fis.OrderDateKey = dd.DateKey
WHERE dd.DateKey=20110101
GROUP BY dd.DateKey

```

Table 17-21. A chasm trap inflates the true values

DateKey	ResellerSalesAmount	InternetSalesAmount
20110101	3,076,816.6244	5,617,883.90

Both the ResellerSalesAmount and InternetSalesAmount in the last query are higher than in the first two queries—and the results in the last query are wrong. The amount of 3,076,816.6244 for the ResellerSalesAmount is double the true amount, as the true amount was multiplied by the number of rows in the FactInternetSales table. The shown amount of 5,617,883.90 results from the true amount of 7,156.54 being multiplied by 785 (the number of rows from the FactResellerSales table).

To overcome this problem, you need to make sure that you split the necessary joins into separate queries. Later, you can then either UNION the two queries (to two rows per date, one for the FactResellerSales and one for the FactInternetSales) or JOIN the two query results (on the now unique DateKey) (see Table 17-22). The first piece of code shows the solution with UNION ALL:

```

SELECT
    DateKey, SUM(SalesAmount) SalesAmount, SUM(SalesCount) SalesCount
FROM
(
    SELECT
        dd.DateKey, SUM(frs.SalesAmount) SalesAmount, COUNT(*) SalesCount
    FROM
        dbo.FactResellerSales frs
    JOIN dbo.DimDate dd ON dd.DateKey = frs.OrderDateKey
    WHERE dd.DateKey=20110101
    GROUP BY dd.DateKey

    UNION ALL

    SELECT
        dd.DateKey, SUM(fis.SalesAmount) SalesAmount, COUNT(*) SalesCount
    FROM
        dbo.FactInternetSales fis
    JOIN dbo.DimDate dd ON dd.DateKey = fis.OrderDateKey
    WHERE dd.DateKey=20110101
    GROUP BY dd.DateKey
) x
GROUP BY DateKey

```

Table 17-22. “UNIONing” the two separate queries overcomes the problem

DateKey	SalesAmount	SalesCount
20110101	1,545,564.8522	787

Alternatively, you can just join the two queries (see also Table 17-23):

```
SELECT
    frs.DateKey,
    frs.ResellerSalesAmount + irs.InternetSalesAmount SalesAmount,
    frs.ResellerSalesCount + irs.InternetSalesCount SalesCount
FROM
(
    SELECT
        dd.DateKey, SUM(frs.SalesAmount) ResellerSalesAmount,
        COUNT(*) ResellerSalesCount
    FROM
        dbo.FactResellerSales frs
    JOIN dbo.DimDate dd ON dd.DateKey = frs.OrderDateKey
    WHERE dd.DateKey=20110101
    GROUP BY dd.DateKey
) frs
JOIN
(
    SELECT
        dd.DateKey, SUM(fis.SalesAmount) InternetSalesAmount,
        COUNT(*) InternetSalesCount
    FROM
        dbo.FactInternetSales fis
    JOIN dbo.DimDate dd ON dd.DateKey = fis.OrderDateKey
    WHERE dd.DateKey=20110101
    GROUP BY dd.DateKey
) irs ON irs.DateKey = frs.DateKey
```

Table 17-23. Joining the two separate queries overcomes the problem

DateKey	SalesAmount	SalesCount
20110101	1,545,564.8522	787

In my demo system, the execution plans for both versions are slightly different, but from a query cost perspective, they are identical.

Fan trap

In the following query, the freight cost (stored in in table SalesOrderHeader) “fans out” to every row of the SalesOrderDetail table. Adding the order’s freight up per order line does lead to a wrong (too high) amount of freight (as it is the freight times the number of rows in the SalesOrderDetail table for that order). Let’s look at the

true values first for the order with SalesOrderID of 43659 (see also Tables 17-24 and 17-25):

```
SELECT
    soh.SalesOrderID, SUM(soh.Freight) Freight, COUNT(*) HeaderCount
FROM
    Sales.SalesOrderHeader soh
WHERE soh.SalesOrderID = 43659
GROUP BY soh.SalesOrderID
```

Table 17-24. The total freight costs of an order

SalesOrderID	Freight	HeaderCount
43659	616,0984	1

```
SELECT
    soh.SalesOrderID, SUM(sod.OrderQty) OrderQty, COUNT(*) DetailCount
FROM
    Sales.SalesOrderHeader soh
JOIN Sales.SalesOrderDetail sod ON sod.SalesOrderID = soh.SalesOrderID
WHERE soh.SalesOrderID = 43659
GROUP BY soh.SalesOrderID
```

Table 17-25. The order has 12 line items

SalesOrderID	OrderQty	DetailCount
43659	26	12

When I combine the two queries naively, I am bitten by the fan trap; the true freight cost of 616.0984 for the particular order is multiplied by the number of order detail rows (12), and a wrong freight amount of 7,393.1808 is returned by the query (see also Table 17-26):

```
SELECT
    soh.SalesOrderID, SUM(soh.Freight) Freight, SUM(sod.OrderQty) OrderQty
FROM
    Sales.SalesOrderHeader soh
JOIN Sales.SalesOrderDetail sod ON sod.SalesOrderID = soh.SalesOrderID
WHERE soh.SalesOrderID = 43659
GROUP BY soh.SalesOrderID
```

Table 17-26. The order header rows are “fanned out” over the order detail rows

SalesOrderID	Freight	OrderQty
43659	7,393.1808	26

Again, you need to split the single query into two, to avoid this effect and get the correct value instead. The first query calculates the freight (without the problematic join

to the SalesOrderDetail table). The second query calculates only the OrderQty by joining the two tables:

```
SELECT
    soh.SalesOrderID, SUM(soh.Freight) Freight, SUM(sod.OrderQty) OrderQty
FROM
    Sales.SalesOrderHeader soh
JOIN (
    SELECT
        soh.SalesOrderID, SUM(sod.OrderQty) OrderQty, COUNT(*) DetailCount
    FROM
        Sales.SalesOrderHeader soh
    JOIN Sales.SalesOrderDetail sod ON sod.SalesOrderID = soh.SalesOrderID
    GROUP BY soh.SalesOrderID
) sod ON sod.SalesOrderID = soh.SalesOrderID
WHERE soh.SalesOrderID = 43659
GROUP BY soh.SalesOrderID
```

Table 17-27. The separated queries are joined into one

SalesOrderID	Freight	OrderQty
43659	7,393.1808	26

Before you start writing a query, you should always look out for join path problems. One way to find out is to look at the entity relationship diagram. In the next section, I show you how you can create one in SQL Server Management Studio.

Entity Relationship Diagrams

SQL Server Management Studio lets you create an ERD for your existing tables (and foreign key constraints). If you expand the database name in the *Object explorer*, you will find *Database Diagrams*. There, you can create a new diagram by adding and arranging tables, or view an existing one. In [Figure 17-1](#), you see an example I created for the *AdventureWorksDW* database. It illustrates the many-to-one relationships between the FactResellerSales table and the date, sales territory, and product dimensions. You can clearly see that this is a snowflake schema, as the DimProduct table has a relationship to DimProductSubcategory, which has a relationship to Dim ProductCategory.



The outcome of `TRUNCATE TABLE demo.Sales` is identical to `DELETE FROM demo.Sales`. There is a difference under the hood, though: `TRUNCATE TABLE` is considered a metadata operation. This operation will not touch a single row of the table but just change the information for the whole table to inform everybody that the table is empty. As you can imagine, this operation is much faster compared to actually deleting every row in a table (as the `DELETE` keyword does). That's why I prefer `TRUNCATE TABLE` over `DELETE`.

You need to consider two things, though. First, `TRUNCATE TABLE` is only possible if the table is not part of any foreign key constraint. If it is, first, you need to `DROP` the constraint, then `TRUNCATE` the table, and then `CREATE` the constraint again. Second, `TRUNCATE TABLE` does not accept any filter conditions, as `DELETE` does via the `WHERE` clause. You can only empty a full table (or partition it, to be precise; “[Partitioning](#)” on page 421 talks about partitions).

The bigger the amount of data gets (as the data grows in size) or the smaller the maintenance windows become (as there are more and more processes running in the maintenance window or because you need to load the data more often, for example multiple times a day), the more critical it is to update your data warehouse more quickly. That's the time when you should think about changing your ETL from a full to a *delta load*. A delta load is more complex, in the sense that it needs to identify the rows that:

- Are new in the data source and therefore need to be inserted into the data warehouse.
- Have changed since the last load and therefore need to be updated in the data warehouse.
- Have been deleted in the data source and therefore need to be either removed from the data warehouse (hard delete) or marked as deleted (soft delete) in the data warehouse.

The following example inserts only rows for new products (products that didn't have a row in the table `Sales` yet). The `WHERE` clause filters for rows that “`NOT EXISTS`” in the data warehouse:

```
-- Delta Load
INSERT INTO demo.Sales (ProductID, SalesAmount)
SELECT
    ISNULL(p.ID, -1) ProductID,
    ps.SalesAmount
FROM
    demo.ProductSales ps
LEFT JOIN demo.Product p ON p.Product=ps.Product
```



```
WHERE NOT EXISTS (SELECT TOP 1 1 FROM demo.Sales ss WHERE ss.ProductID =  
ISNULL(p.ID, -1))
```

In case already-loaded data can change as well, you need to add an UPDATE statement. And in case already-loaded data can be deleted in the data source, you need to add a DELETE statement (for a hard delete) or an UPDATE statement (to implement a soft delete) as well. More often than not, the business users want to keep track of such changes: when a row was inserted, updated or modified and what the row looked like before the modification? In [“Slowly Changing Dimensions” on page 387](#), I show how to add such information by implementing slowly changing dimensions.

Keep in mind that a delta load will only be faster if the physical organization (index strategy, table partitioning) of the tables in the data source and in the data warehouse supports the JOIN predicates and the WHERE conditions. If not, the delta load might take as long as a full load (or even longer due to added filter conditions). In [Chapter 20](#), you will learn which indexes you should create and how you can manage partitions in a relational table.

To further refresh your data model in Power BI or Analysis Services more quickly, you need to create partitions there and in the data warehouse. Usually, you partition your tables time-wise (e.g., one partition per day). A partition stores rows of a table together. In a relational database, you can quickly truncate the content of a partition before you reload the data for it. In Power BI and Analysis Services, you can specify that only a partition should be refreshed, instead of the whole table. This will speed up the refresh time tremendously, since you won’t need to refresh all the (certainly unchanged) old data from years ago over and over again. It is important that the partitions are aligned between the relational database and Power BI/Analysis Services, to get the best performance.

With this section about ETL, the first chapter about SQL is finished.

Key Takeaways

In this chapter, I introduced the moving parts of a relational database that is queried and maintained via SQL. Here are some of the main points:

- You learned about the basic parts of a relational database: tables, columns, relationships, primary keys, and foreign keys.
- I recommended creating a dedicated schema for all objects you want to expose to Power BI, or Analysis Services (or any other reporting layer), and to which you give read-only access to all tools and people who need access.
- The content of the data warehouse can be exposed as tables, views, stored procedures, or functions. Views are the most common way and are a good starting point. In case of performance issues when querying the content, the information

can be persisted as a table. Implementing the queries as stored procedures or as functions makes it less user friendly for the data consumers, but allows for parameters and optimized T-SQL code, where necessary.

- There are tools available to visualize the tables and their foreign key constraints as an entity relationship diagram. Foreign key constraints restrict manipulations of the content of the tables (INSERT, UPDATE, and DELETE), but have no direct influence on SELECT.
- You must define the kind of relationship in each and every query (JOIN operator in SELECT statement)—they are not automatically added behind the scenes for you. On the one hand, this gives you full control and flexibility about how you combine the contents of tables. But be aware of traps in your data model that might lead to too few or too many rows in the query result (or too low or too high numbers in aggregated values).
- In SQL Server Management Studio, you can easily create an entity relationship diagram and get a quick overview about tables and their relationships, which will allow you to discover potential join path problems.
- If you need to keep track of changes to dimensions, you can implement slowly changing dimensions in a relational data warehouse during the ETL process.

Now that you've met your new best friends in the world of SQL, it's about time to learn what these friends can do for you for you in terms of building a data model that will help you in your adventures in Power BI.

Building a Data Model with SQL

SQL as a language is quite mature, and if you master this language, you can make transformations of your tables very easy. Similar to some applied steps in Power Query, *dynamic* SQL can make our life even easier as it can make your code resilient to changes. Instead of manually maintaining code, you can write code to maintain the solution for you in an (semi-)automatic way.

After many years working with analytic solutions, I still believe that a data warehouse is the powerhouse in a business intelligence architecture. Putting transformations in a relational layer (which does not necessarily persist all the content) opens your solution to many tools. Of course, Power BI (via Power Query) can consume data from a relational database. But also, plenty of other tools (and users) will be able to connect to a relational database. In a scenario where you or your colleagues have—God forbid!—BI and reporting tools other than Power BI in place, a relational data warehouse (layer) can be the common ground and important puzzle piece to achieve the single source of truth. Instead of re-implementing transformations in all those tools, you can do them in the data warehouse.

All solutions around SQL can be accomplished in different ways:

Persist the content into a table

This is practical if tools or users query content of the data warehouse regularly (e.g., if you use Power BI Report Builder on top of the relational database instead of a Power BI semantic model, or if you run Power BI in *DirectQuery* mode). You can create indexes and partitions on this table to speed up the queries even more.

Persist a query only as a view

This is practical if the sole purpose of the relational database is to provide all transformations but the data is cached in a different place (e.g., if you load the

data into Power BI and run it in *Import* mode). If the data is only read once per refresh, persisting the data usually does not pay off.

Use the indexed or materialized view feature

Many relational database management systems have a feature called *indexed view* or *materialized view*. Through this, the result of a *view* is persisted and offers similar performance to that of an indexed table. The advantage, though, is that you do not need setup schedules to refresh the content of the indexed view—the database management system will make sure to keep it updated whenever the underlying table(s) are manipulated. The disadvantage is that it will slow down the manipulation of the underlying tables (due to the extra steps needed for updating the indexed view).

Create functions and/or stored procedures

You can pass parameters into functions and stored procedures, which can make handling them easier for users (and you can hide complex logic reacts in different ways to the parameters from them).



I strongly recommend giving the report creators access only to views (and maybe functions or stored procedures) and not the data warehouse tables. Put all those artifacts into a dedicated database schema and give read access to this schema only (and no other schema). In [Chapter 20](#), I will discuss the different options from a performance perspective.

Normalizing

The goal of normalizing is to move redundant information into a different table and only keep a foreign key to reference the row in the (now) separate table. Normalizing in SQL means that you list the columns you want in the projection (by writing them right after the `SELECT` keyword). The question is, again, how to decide which columns you want to keep in a result set. In SQL, you can find dimension candidates by using `COUNT` and `GROUP BY`. By counting the number of rows per distinct value of a column, you can easily discover redundant information.¹

The table `demo.financials` in my example contains several columns:

- Segment
- Country
- Product
- Discount Band
- Units Sold
- Manufacturing Price

¹ The file used for this section's demonstrations is [201 Normalizing.sql](#).

- Sale Price
- Gross Sales
- Discounts
- Sales
- COGS
- Profit
- Date
- Month Number
- Month Name
- Year

A quick glance will identify Units Sold, Sale Price, Gross Sales, Discounts, Sales, COGS, Profit as the facts: they are numeric and aggregable (except for the Sale Price). The other columns (including all date-related columns like month and year) are candidates for normalization into their own table.

From the following query, you learn that there are 700 rows in total in the table but only five segments:

```
SELECT COUNT(*) CountAll, COUNT(DISTINCT Segment) CountSegment
FROM demo.financials;
```

CountAll	CountSegment
700	5

The Segment is a redundant column in the financials table. Therefore, I remove it from financials and create a separate Segment table:

```
CREATE TABLE normalization.Segment (
    SegmentID int IDENTITY(1,1) PRIMARY KEY,
    Segment nvarchar(50)
)
```

Column SegmentID is the PRIMARY KEY of this new table. The content for each row of this column is automatically determined by the database (IDENTITY(1,1)). The only “real” content (which will later be exposed to the report creators) is the column Segment.

During the INSERT statement, you need to make sure to specify the DISTINCT keyword (so each unique Segment is only inserted once). A mistake here would result in a later duplication of the rows in the fact table:

```
INSERT INTO normalization.Segment
SELECT
    DISTINCT
    Segment
FROM
    demo.financials;
```

Additionally, I strongly recommend that you insert a placeholder value, in case a fact row contains an invalid segment or for when the segment should contain NULL. It's best practice to have a row with value -1 for the ID column in a dimension table (which will be later referenced when the rows are inserted into the fact table):

```
SET IDENTITY_INSERT normalization.Segment ON;  
INSERT INTO normalization.Segment (SegmentID, Segment) VALUES (-1, 'unknown');  
SET IDENTITY_INSERT normalization.Segment OFF;
```

Before you can directly set the value for an identity column, you need to set the IDENTITY_INSERT property of the table to ON. Don't forget to set it to OFF right afterward to avoid the identity column being used in other INSERT or UPDATE statements by mistake. The provided value for the Segment column ("unknown" in the preceding example) should be something that is meaningful to report consumers. I will talk about your options again in ["Flags and Indicators" on page 379](#).

When you count the distinct values of the Product and the [Manufacturing Price] column, you will discover that they have the same count:

```
SELECT COUNT(*) CountAll, COUNT(DISTINCT Product) CountProduct,  
COUNT(DISTINCT [Manufacturing Price]) CountManufacturingPrice  
FROM demo.financials;
```

CountAll	CountProduct	CountManufacturingPrice
700	6	6

As it turns out, this is not by chance but because a transitive dependency exists between the two columns: there is exactly one [Manufacturing Price] per Product:

```
SELECT  
    DISTINCT  
        Product,  
        [Manufacturing Price]  
FROM  
    demo.financials;
```

Product	Manufacturing Price
Amarilla	260.00
Carretera	3.00
Montana	5.00
Paseo	10.00
Velo	120.00
VTT	250.00

That's why I will put them together into the same dimension table.

You need to repeat these steps for all dimensional attributes of the table `financials`:

- Segment (Segment)
- Country (Country)
- Product (Product, [Manufacturing Price])
- Discount Band ([Discount Band])

The date-related columns are special. They are definitely candidates for a dimension table, but you cannot derive its content from the table `financials`. The reason is that Power BI will need a table that contains a row for every day of a calendar year. The rows in the table `financials` violate this condition, as you can see here:

```
SELECT
    COUNT(*) CountAll,
    MIN([Date]) MinDate,
    MAX([Date]) MaxDate,
    COUNT(DISTINCT [Date]) CountDay
FROM
    demo.financials;
```

CountAll	MinDate	MaxDate	CountDay
700	2013-09-01	2014-12-01	16

The range of dates is between September 1, 2013 and December 1, 2014 and therefore does not cover a full calendar year. On top of that, it only contains 16 different dates. This is a clear violation of the rules for a proper date table (at least when it comes to Power BI). This is actually pretty typical for a fact table, as transactions will not happen every day; also, fact tables are often already aggregated (like in this case, where the facts are aggregated to the first of the month). That's why you should never derive the content for the date dimension from the fact table. I show how to create a date table properly in [“Time and Date” on page 383](#).

After every dimension has its own table, I can finally normalize the fact table (in a schema named `normalization`). This table will only contain numerical columns—integers for the foreign keys and decimals for the facts:

```
CREATE TABLE normalization.financials (
    SegmentID int NOT NULL
        CONSTRAINT FK_financials_Segment
        REFERENCES normalization.Segment(SegmentID),
    CountryID int NOT NULL
        CONSTRAINT FK_financials_Country
        REFERENCES normalization.Country(CountryID),
    ProductID int NOT NULL
        CONSTRAINT FK_financials_Product
        REFERENCES normalization.Product(ProductID),
```

```

DiscountBandID int NOT NULL
                CONSTRAINT FK_financials_DiscountBand
                REFERENCES normalization.[Discount Band](DiscountBandID),
DateKey int,
[Units Sold] decimal(18,2),
[Sales Price] decimal(18,2),
[Gross Sales] decimal(18,2),
[Discounts] decimal(18,2),
[Sales] decimal(18,2),
[COGS] decimal(18,2),
[Profit] decimal(18,2)
)

```

I do not allow NULL as a value for a foreign key. To tell the database system that these columns are indeed foreign key columns, I add the CONSTRAINT keyword to give the constraint a proper name (which will make it easier to drop or disable the constraint later, if needed) and specify the dimension table and its primary key column after REFERENCES. As I have not created the Date table, I omit the foreign key constraint here.

In **Example 18-1**, I then insert all rows from financials into the newly created table.

Example 18-1. Inserting rows in the newly created table

```

INSERT INTO normalization.financials
SELECT
    ISNULL(s.SegmentID, -1) as SegmentID,
    ISNULL(c.CountryID, -1) as CountryID,
    ISNULL(p.ProductID, -1) as ProductID,
    ISNULL(d.DiscountBandID, -1) as DiscountBandID,
    YEAR(f.[Date]) * 10000 + MONTH(f.[Date]) * 100 + DAY(f.[Date])
    as DateKey,
    f.[Units Sold],
    f.[Sale Price],
    f.[Gross Sales],
    f.[Discounts],
    f.[Sales],
    f.[COGS],
    f.[Profit]
FROM
    demo.financials f
LEFT JOIN normalization.Segment s ON
    s.Segment = f.Segment
LEFT JOIN normalization.Country c ON
    c.Country = f.Country
LEFT JOIN normalization.Product p ON
    p.Product = f.Product
LEFT JOIN normalization.[Discount Band] d ON
    d.[Discount Band] = f.[Discount Band]

```


For the values of the foreign key columns, I added a safeguard in cases where the fact table contains a business key (e.g., `Segment`), which is not (yet) inserted into the dimension table. If you run the queries in the exact order discussed here (first the insert statements for the dimension, and in the end the insert statements for the fact table), then this will not be necessary. I tend to add this logic anyways, as I prefer being safe over being sorry, for cases where the order may have been mixed up for some reason. The safeguard is twofold:

- I intentionally use a `LEFT JOIN` (instead of an inner join).
- I replace `NULL` values (via function `ISNULL`) with a value of `-1`.

Without this safeguard, either rows from the fact table could be filtered out (by an inner join), or a value of `NULL` would be inserted into the foreign key columns. Both are bad; if you unintentionally filter out rows from the fact table, the content of your data warehouse would be wrong. If you add fact rows with `NULL` as their foreign key, every report would need to use a (slower) `LEFT JOIN` rather than a (faster) `INNER JOIN`. I prefer the latter. The previously inserted extra row in the dimension tables (with `-1` as the primary key), and the safeguard here will allow you to always do `INNER JOINs` between the tables in the data warehouse. And even missing or wrong dimensions do not lead to empty descriptions shown in the report, but to meaningful text (e.g., “unknown”).

The value for the `DateKey` foreign key column is derived from the fact’s actual date: the date’s year is multiplied by 10,000 (e.g., 20,230,000), then the month number is multiplied by 100 (e.g., 800) and added to the year number, and finally the day number of the month is added as well (for a total of 20,230,801). Despite the rule that a surrogate key in a data warehouse should always be considered meaningless, it is best practice to allow the date table’s key an exception. This key is still readable. Nevertheless, in Power BI, you should never use the fact table’s foreign key to filter the rows, but always join the date dimension and put filters there, as I explained in [“Normalizing and Denormalizing” on page 109](#).

I wouldn’t recommend directly using the query from [Example 18-1](#) when connecting Power BI to the data source but instead provide the result in the database. As mentioned in [“Normalizing” on page 44](#), you have several options to do so: persist into a table, create a view, create a function, or create a procedure.

Persisting into a Table

For the dimension tables, there is no way around persisting if you want to use the `IDENTITY` feature. This is only available for rows physically persisted in a table. As a first step, you need to drop all foreign key constraints referencing the table. Then, you drop the table if it already exists. Finally, you (re-)create the table (in the right

schema) and insert the rows. Don't forget to insert the extra row with primary key -1. For the Segment table, this looks like this:

```
-- Segment
IF EXISTS (
    SELECT TOP 1 1
    FROM sys.objects
    WHERE OBJECT_NAME(object_id) = 'FK_financials_Segment'
)
ALTER TABLE normalization.financials
DROP CONSTRAINT FK_financials_Segment;

DROP TABLE IF EXISTS normalization.Segment;

CREATE TABLE normalization.Segment (
    SegmentID int IDENTITY(1,1) PRIMARY KEY,
    Segment nvarchar(50)
);
GO

INSERT INTO normalization.Segment
SELECT
    DISTINCT
    Segment
FROM
    demo.financials;

SET IDENTITY_INSERT normalization.Segment ON;
INSERT INTO normalization.Segment (SegmentID, Segment)
VALUES (-1, 'unknown');
SET IDENTITY_INSERT normalization.Segment OFF;
```

For the fact table(s), the steps are simpler: you only need to drop the table (if it exists) and then insert all the rows. No need to take extra care of constraints or the extra row (-1):

```
-- Financials
DROP TABLE IF EXISTS normalization.Financials;
CREATE TABLE normalization.Financials (
    SegmentID int NOT NULL
        CONSTRAINT FK_Financials_Segment
        REFERENCES normalization.Segment(SegmentID),
    CountryID int NOT NULL
        CONSTRAINT FK_Financials_Country
        REFERENCES normalization.Country(CountryID),
    ProductID int NOT NULL
        CONSTRAINT FK_Financials_Product
        REFERENCES normalization.Product(ProductID),
    DiscountBandID int NOT NULL
        CONSTRAINT FK_Financials_DiscountBand
        REFERENCES normalization.[Discount Band]
        (DiscountBandID),
```

```

        DateKey int,
        [Units Sold] decimal(18,2),
        [Sales Price] decimal(18,2),
        [Gross Sales] decimal(18,2),
        [Discounts] decimal(18,2),
        [Sales] decimal(18,2),
        [COGS] decimal(18,2),
        [Profit] decimal(18,2)
    );
GO
INSERT INTO normalization.Financials
SELECT
    ISNULL(s.SegmentID, -1) as SegmentID,
    ISNULL(c.CountryID, -1) as CountryID,
    ISNULL(p.ProductID, -1) as ProductID,
    ISNULL(d.DiscountBandID, -1) as DiscountBandID,
    YEAR(f.[Date]) * 10000 + MONTH(f.[Date]) * 100 + DAY(f.[Date])
    as DateKey,
    f.[Units Sold],
    f.[Sale Price],
    f.[Gross Sales],
    f.[Discounts],
    f.[Sales],
    f.[COGS],
    f.[Profit]
FROM
    demo.financials f
LEFT JOIN normalization.Segment s ON
    s.Segment = f.Segment
LEFT JOIN normalization.Country c ON
    c.Country = f.Country
LEFT JOIN normalization.Product p ON
    p.Product= f.Product
LEFT JOIN normalization.[Discount Band] d ON
    d.[Discount Band]= f.[Discount Band]

```

Creating a View

Instead of duplicating all the data and persisting it into a table, on many occasions, a view will be enough (e.g., when you do not need to create a surrogate key or when a query is not used to actually query per se but is only used to refresh a Power BI semantic model or an Analysis Services database). A view can be described as a “virtual” table. No data is duplicated; only the SELECT statement is stored in the view’s definition. As I already created a table with the name `Financials` in the `normalization` schema, I decided to add `vw_` as a prefix to the view’s name. The data types for the columns of the view are automatically derived from the columns referenced in the view:

```

CREATE OR ALTER VIEW normalization.vw_Financials AS (
SELECT

```

```

        ISNULL(s.SegmentID, -1) as SegmentID,
        ISNULL(c.CountryID, -1) as CountryID,
        ISNULL(p.ProductID, -1) as ProductID,
        ISNULL(d.DiscountBandID, -1) as DiscountBandID,
        YEAR(f.[Date]) * 10000 + MONTH(f.[Date]) * 100 + DAY(f.[Date])
        as DateKey,
        f.[Units Sold],
        f.[Sale Price],
        f.[Gross Sales],
        f.[Discounts],
        f.[Sales],
        f.[COGS],
        f.[Profit]
FROM
    demo.financials f
LEFT JOIN normalization.Segment s ON
    s.Segment = f.Segment
LEFT JOIN normalization.Country c ON
    c.Country = f.Country
LEFT JOIN normalization.Product p ON
    p.Product= f.Product
LEFT JOIN normalization.[Discount Band] d ON
    d.[Discount Band]= f.[Discount Band]
)

```

The view can be queried the same way as you would query a table. Either with a SELECT statement, or simply in Power Query by selecting its name in the list of “tables” (which also exposes the view, just with a slightly different icon in front of its name):

```
SELECT * FROM normalization.vw_Financials;
```

Creating a Function

Typically, you would create a function (instead of a view) because you want to add parameters you pass into the query. I therefore sometimes refer to (table-valued) functions as “parametrized views.” To demonstrate this, I add the parameter Date Key, which is used as a filter in the WHERE clause of the query. Again, I use a prefix (fn_) to distinguish it as a database object from the table and the view:

```

-- FUNCTION
CREATE OR ALTER FUNCTION normalization.fn_Financials (
    @Date date
)
RETURNS TABLE
AS
RETURN
SELECT
    ISNULL(s.SegmentID, -1) as SegmentID,
    ISNULL(c.CountryID, -1) as CountryID,
    ISNULL(p.ProductID, -1) as ProductID,

```

```

        ISNULL(d.DiscountBandID, -1) as DiscountBandID,
        YEAR(f.[Date]) * 10000 + MONTH(f.[Date]) * 100 + DAY(f.[Date])
    as DateKey,
    f.[Units Sold],
    f.[Sale Price],
    f.[Gross Sales],
    f.[Discounts],
    f.[Sales],
    f.[COGS],
    f.[Profit]
FROM
    demo.financials f
LEFT JOIN normalization.Segment s ON
    s.Segment = f.Segment
LEFT JOIN normalization.Country c ON
    c.Country = f.Country
LEFT JOIN normalization.Product p ON
    p.Product= f.Product
LEFT JOIN normalization.[Discount Band] d ON
    d.[Discount Band]= f.[Discount Band]
WHERE [Date] = ISNULL(@Date, [Date]);

```

The only difference from view (beyond the differences in the keywords you need to use to define a view versus a function) is the added line for the WHERE clause. I use the function ISNULL to fail over the row's [Date] value. This allows you to use NULL for the @Date parameter. When this is the case, then all fact rows are returned (instead of none). This makes the parameter optional, as you can see in the following two examples.

Unfortunately, Power Query will not expose functions to you. Instead, providing a SQL query as the data source is mandatory. You can use this function in a similar way to a table or a view in a SELECT statement. Don't forget to add opening and closing parentheses after the function's name, though. In the first example, the parameter is respected and 35 rows that match the parameter value are returned:

```

SELECT * FROM normalization.fn_Financials ({d'2014-01-01'});

```

On the other hand, if I provide NULL as the parameter value, all 700 rows are returned:

```

SELECT * FROM normalization.fn_Financials (null);

```

Creating a Procedure

To make this list of options complete, I'm adding a definition for a stored procedure. Again, the example contains the query and a parameter for the date column. Typically, you would create a stored procedure because you want or need to add more logic. A procedure can use all of the good stuff the procedural extension (called

T-SQL, for transactional SQL) offers, including parameters. Again, I use a prefix (“usp_”) for the procedure’s name:

```
-- PROCEDURE
CREATE OR ALTER PROCEDURE normalization.usp_Financials (
    @Date date = null
)
AS
SELECT
    ISNULL(s.SegmentID, -1) as SegmentID,
    ISNULL(c.CountryID, -1) as CountryID,
    ISNULL(p.ProductID, -1) as ProductID,
    ISNULL(d.DiscountBandID, -1) as DiscountBandID,
    YEAR(f.[Date]) * 10000 + MONTH(f.[Date]) * 100 + DAY(f.[Date])
    as DateKey,
    f.[Units Sold],
    f.[Sale Price],
    f.[Gross Sales],
    f.[Discounts],
    f.[Sales],
    f.[COGS],
    f.[Profit]
FROM
    demo.financials f
LEFT JOIN normalization.Segment s ON
    s.Segment = f.Segment
LEFT JOIN normalization.Country c ON
    c.Country = f.Country
LEFT JOIN normalization.Product p ON
    p.Product = f.Product
LEFT JOIN normalization.[Discount Band] d ON
    d.[Discount Band] = f.[Discount Band]
WHERE [Date] = ISNULL(@Date, [Date]);
```

Unfortunately, procedures cannot be used as part of a the FROM or JOIN clause, as you are using tables, views, and functions. You need to call a procedure via the EXEC command instead. This makes stored procedures a less common way of exposing information from the data warehouse. Before you decide to create procedures, make sure that the reporting tool you want to use can handle procedures. In Power BI, you would connect to the database and specify a *query* as the data source.

```
EXEC normalization.usp_Financials {d'2014-01-01'};
```

As I defined the procedure’s parameter with a default value (@Date date = null), you can also omit the parameter value if you want to have it contain the value NULL:

```
EXEC normalization.usp_Financials null;
```

```
EXEC normalization.usp_Financials;
```

In the rest of **Part V**, I will show the query but won't repeat the code to persist the query as a table, or transform it into a view, function, or procedure, for the sake of brevity.

Creating a Filter Dimension

I also want to demonstrate the solution of a *Junk* or *Filter* dimension in SQL (as described in “**Normalizing and Denormalizing**” on page 109). Especially when the dimensions do not have many attributes (or just one, as is the case for *Segment*, *Country*, or *Discount Amount*), this approach can reduce storage space, as the fact table only needs to contain a single foreign key (the composite business key), instead of several independent foreign keys.

First, you create the *Filter* table:

```
CREATE TABLE normalization_filter.[Filter]
(
    _FilterKey [int] IDENTITY(1, 1) PRIMARY KEY,
    [Segment] [nvarchar](50) NULL,
    [Country] [nvarchar](50) NULL,
    [Product] [nvarchar](50) NULL,
    [Discount Band] [nvarchar](50) NULL,
    [Manufacturing Price] [decimal](18, 2) NULL
) ON [PRIMARY];
```

When I physically create a table (instead of just creating a view), I will definitely add a surrogate key (*_FilterKey*).

The table will be populated with a distinct list of possible combinations of the dimensional values, derived from the *financials* table. Column *_FilterKey* is not part of the list, as it will be auto-populated due to the *IDENTITY(1, 1)* clause used during the *CREATE TABLE* statement.

```
INSERT INTO normalization_filter.[Filter]
SELECT
    DISTINCT
        [Segment],
        [Country],
        [Product],
        [Discount Band],
        [Manufacturing Price]
FROM
    demo.financials;
```

There's nothing special about the fact table. It has columns for the *_FilterKey* and all facts:

```
CREATE TABLE normalization_filter.[Financials]
(
    [_FilterKey] [int] NULL
```

```

REFERENCES normalization_filter.[Filter] (_FilterKey),
[Units Sold] [decimal](18, 2) NULL,
[Sale Price] [decimal](18, 2) NULL,
[Gross Sales] [decimal](18, 2) NULL,
[Discounts] [decimal](18, 2) NULL,
[Sales] [decimal](18, 2) NULL,
[COGS] [decimal](18, 2) NULL,
[Profit] [decimal](18, 2) NULL,
[Date] [date] NULL
) ON [PRIMARY];

```

The SELECT statement for the INSERT operation queries all rows from the table `financials`. To insert the correct `_FilterKey` value, a lookup to the newly created dimension table (`Filter`) is necessary. The JOIN predicate includes all columns from the `Filter` table. In theory, you could omit the `Manufacturing Price`, as this column is transitive dependent on the `Product` column. In practice, I want to make sure that my code also covers situations where a product suddenly comes with different prices. The same is true for the LEFT JOIN. As I just have inserted a distinct list of possible combinations into the `Filter` table, I don't want to risk losing any rows from the fact table in case there was a problem when the `Filter` table was populated. The LEFT JOIN guarantees that no rows from the fact table can be lost. Here's the code:

```

INSERT INTO normalization_filter.[Financials]
SELECT
    d._FilterKey
    ,f.[Units Sold]
    ,f.[Sale Price]
    ,f.[Gross Sales]
    ,f.[Discounts]
    ,f.[Sales]
    ,f.[COGS]
    ,f.[Profit]
    ,f.[Date]
FROM
    [demo].[financials] f
LEFT JOIN
    normalization_filter.[Filter] d ON
        d.[Segment] = f.[Segment] AND
        d.[Country] = f.[Country] AND
        d.[Product] = f.[Product] AND
        d.[Discount Band] = f.[Discount Band] AND
        d.[Manufacturing Price] = f.[Manufacturing Price]

```


Denormalizing

Denormalizing can be seen as the opposite of normalizing. Instead of storing a reference, the goal is to have the actual value(s) in the current table, even when these values are redundant and transitive dependent. If you have a background in IT and databases, this step might be counterintuitive for you. Personally, I had to overcome an inner resistance to introducing redundancy to a table, after the two decades I've spent learning how to avoid redundancies, and practicing doing so, before starting building data warehouses, where redundancies are not only accepted but are best practice for dimension tables.

You can access the columns of the referenced table in a query by joining the referencing and referenced table together. Remember: you need to denormalize dimension tables to achieve a star schema. And while you are transforming a table in SQL, you should add an integer surrogate key to the dimension table. Therefore, I would recommend persisting every dimension as a table, as shown in the following code example for the table Product. Only the primary key (ProductID) is of data type integer (with an IDENTITY specification); the rest of the columns are defined as Unicode strings of variable length (and a maximum of 50 characters, which should fit the expected text):²

```
CREATE TABLE denormalization.Product (  
    ProductID int IDENTITY(1,1) PRIMARY KEY,  
    Product nvarchar(50),  
    Subcategory nvarchar(50),  
    Category nvarchar(50)  
);
```

The content for this table is created via joins from three tables (DimProduct, DimProductSubcategory, and DimCategory):

```
INSERT INTO denormalization.Product  
SELECT  
    dp.EnglishProductName Product,  
    ISNULL(dps.EnglishProductSubcategoryName, 'unknown') Subcategory,  
    ISNULL(dpc.EnglishProductCategoryName, 'unknown') Category  
FROM  
    dbo.DimProduct dp  
LEFT JOIN dbo.DimProductSubcategory dps ON  
    dps.ProductSubcategoryKey=dp.ProductSubcategoryKey  
LEFT JOIN dbo.DimProductCategory dpc ON  
    dpc.ProductCategoryKey=dps.ProductCategoryKey
```

² The file used for this section's examples is *202 Denormalizing.sql*.

After you have created the perfect star schema, as you normalized all fact tables and denormalized all dimension tables, it is time to think about adding information in the form of calculations.



I intentionally use LEFT JOINS in case referential integrity is not guaranteed (if a product's ProductSubcategoryKey can't be found in DimProductSubcategory). On top of that, I recommend finding a good replacement value for the subcategory's name stored in the Product table, like "Unknown." In ["Flags and Indicators" on page 379](#), I show ways to assign replacement values gracefully.

If referential integrity is guaranteed (active foreign key constraints are in place between the tables used in the join), you should use INNER JOINS instead; they're more performant. No treatment for replacement values is necessary, then.

Calculations

SQL offers standard operators and a wide variety of mathematical and statistical function. I won't go into too much detail about them here. You can find [all the operators](#) and [functions](#) explained very well in Microsoft's official online documentation for free.

Before you start creating calculations in SQL, evaluate whether the result will indeed be additive (that means that aggregating the result will lead to a meaningful result). A rule of thumb is that calculations involving a division operator (e.g., to calculate an average, a percentage, or a ratio) are not additive. Those calculations need to be done on the report level (read: on aggregated values and not per individual rows of the table), and therefore be defined as a measure in the data model. You can find typical examples for such calculations in DAX in ["Calculations" on page 190](#).

To demonstrate the problem with non-aggregable calculations, I created the following view, which aggregates and calculates numbers based on the available facts in the FactResellerSales table:³

```
CREATE OR ALTER VIEW calc.SalesAggregation AS (  
  SELECT  
    frs.ProductKey,  
    SUM(frs.OrderQuantity) as OrderQuantity,  
    AVG(frs.UnitPrice) as UnitPrice, -- dangerous  
    SUM(frs.TotalProductCost) as TotalProductCost,  
    SUM(frs.DiscountAmount) as DiscountAmount,  
    SUM(SalesAmount) - SUM(frs.TotalProductCost) as Margin,
```

³ The file used for the examples in this section is [203 Calculations.sql](#).

```

        (SUM(SalesAmount) - SUM(frs.TotalProductCost))/SUM(SalesAmount)
        as MarginPct,
        SUM(frs.SalesAmount) as SalesAmount,
        COUNT(frs.SalesAmount) as SalesAmountCount,
        AVG(frs.SalesAmount) as SalesAmountAvg -- dangerous
FROM
    dbo.FactResellerSales frs
GROUP BY frs.ProductKey
);

```

Based on this view, I calculate the grand margin (a single number calculated over all rows) once as the difference of SalesAmount and TotalProductCost (named Margin1) and once as the sum of the view's Margin column (named Margin2):

```

-- Margin
SELECT
    FORMAT(SUM(SalesAmount) - SUM(TotalProductCost), '#,###') Margin1,
    FORMAT(SUM(Margin), '#,###') Margin2
FROM
    calc.SalesAggregation;

```

Margin1	Margin2
470,483	470,483

The result is exactly the same. And the reason is that it makes no difference if you first add up the values for SalesAmount and TotalProductCost and then subtract the two numbers (as was done in Margin1) or if you first subtract the two values per product and then add up the results (as was done in Margin2). Such a calculation (result of a subtraction) is aggregable and can be implemented either way.

Next, let's look at ways to calculate the sales amount. I implemented five different versions. The first version just sums up the column from the view (SalesAmount1). The second version multiplies the UnitPrice by the OrderQuantity and subtracts the DiscountAmount from the result (SalesAmount2). Alternatively, I calculate the average of the UnitPrice, multiply this by the sum of the OrderQuantity, and then subtract the DiscountAmount (SalesAmount3). The next two versions ignore the view and directly query the base table (FactResellerSales) to avoid the aggregations done in the view needing to be. The sum of the SalesAmount is calculated once (SalesAmount4). For SalesAmount5, I implement the formula per row of Fact ResellerSales and add up the values. The expectation is that the last two versions will be identical and correct.

But what about the other versions? Here you are:

```

-- SalesAmount
SELECT
    FORMAT(SUM(SalesAmount), '#,###') SalesAmount1,
    FORMAT(SUM((UnitPrice * OrderQuantity) - DiscountAmount), '#,###')

```

```

        SalesAmount2,
        FORMAT((AVG(UnitPrice) * SUM(OrderQuantity)) - SUM(DiscountAmount),
        '#,###') SalesAmount3
FROM
    calc.SalesAggregation

SELECT
    FORMAT(SUM(SalesAmount4), '#,###') SalesAmount4,
    FORMAT(SUM(SalesAmount5), '#,###') SalesAmount5
FROM
    (SELECT
        SalesAmount as SalesAmount4,
        (UnitPrice*OrderQuantity)-DiscountAmount SalesAmount5
    FROM dbo.FactResellerSales
    ) x

```

SalesAmount1	SalesAmount2	SalesAmount3	SalesAmount4	SalesAmount5
80,450,597	80,722,815	101,291,559	80,450,597	80,450,597

SalesAmount1, SalesAmount4, and SalesAmount5 are identical, and trust me, they are correct. SalesAmount2 is (slightly) off and therefore wrong. SalesAmount3 is completely off. These two versions have in common that they are based on the (aggregated) UnitPrice from the view SalesAggregation, which is calculated as the average of the UnitPrice per ProductKey. Function AVG calculates only an arithmetic average. That means it sums up all values of UnitPrice (per ProductKey), and then divides the sum by the number of rows (per ProductKey). This would only work correctly if only a single product was sold. The correct calculation must take the OrderQuantity into account.

Imagine you bought apples one day for the price of \$1 per pound and for \$2 per pound another day. If you bought a pound each day, then the average price is \$1.5. But imagine you bought 1 pound for \$1 and 10 pounds for \$2—then you can not safely pretend that the average price was \$1.5 (as the AVG function will calculate). The correct formula has to take the quantity into account: 1 pound × \$1 plus 10 pounds × \$2 = \$21. Dividing this by 11 pounds gives the correct average price of \$1.9.

The problem with SalesAmount2 is that it multiplies the incorrectly calculated UnitPrice from the view with the OrderQuantity. The problem with SalesAmount3 is that it averages the wrongly calculated UnitPrice. The calculation formula of SalesAmount5 works because it is not based on the (aggregated values of the) view but is calculated on the granularity level of the table FactResellerSales.

The correct calculation for the UnitPrice on any aggregation level is as a sum of the SalesAmount divided by the sum of OrderQuantity. This you can only achieve as a calculation on the report level, which means in DAX, when it comes to Power BI.

MarginPct is the Margin in percent of the SalesAmount. To calculate “in percent” means that the Margin has to be divided by the SalesAmount. The following query shows the Margin and the SalesAmount and then two versions for the calculation of the margin in percent: one averages the MarginPct from the view (which calculates the margin in percent per ProductKey), and the other calculates it as the division of the sum of Margin and sum of SalesAmount:

```
-- MarginPct
SELECT
    FORMAT(SUM(Margin), '#,###') Margin,
    FORMAT(SUM(SalesAmount), '#,###') SalesAmount,
    FORMAT(AVG(MarginPct), '0.00%') MarginPct1,
    FORMAT(SUM(Margin)/SUM(SalesAmount), '0.00%') MarginPct2
FROM
    calc.SalesAggregation;
```

Margin	SalesAmount	MarginPct1	MarginPct2
470,483	80,450,597	8.34%	0.58%

The numbers for the two versions are very different. Averaging the MarginPct, as in the calculation for MarginPct1, does not lead to the correct number. The only way is to divide the two aggregated (and shown values): 470,483 divided by 80,450,597 leads to the correct value of 0.58%.



It is not enough that the numbers in the tables and views provided in the data warehouse are correct. You need to think beyond the data warehouse and decide if a calculation’s result is aggregable, if the result is additive. If it is not, then pre-calculating the value in the data warehouse layer makes no sense. In fact, it could lead to wrongly reported numbers, if somebody aggregates the values in visuals and reports.

Flags and Indicators

Remember: the goal of all the exercises in transforming source data is to make the lives of report creators easier. Usually, showing codes or abbreviations in a report is not what the report consumer wants. SQL is good central place to transform such flags and indicators into meaningful text:⁴

⁴ This section uses for *204 Flags and Indicators.sql* for the examples.

- To transform the column FinishedGoods (which contains 0 or 1) to a descriptive text column, use the CASE operator and provide different conditions to convert the flag into not salable, salable, or unknown:

```
SELECT
    FinishedGoodsFlag as _FinishedGoodsFlag,
    CASE
        WHEN FinishedGoodsFlag=0 THEN 'not salable'
        WHEN FinishedGoodsFlag=1 THEN 'salable'
        ELSE 'unknown'
    END [Finished Goods Flag]
,*
FROM dbo.DimProduct;
```

_FinishedGoodsFlag	Finished Goods Flag	ProductKey	ProductAlternateKey	...
...
0	not salable	208	TP-0923	...
0	not salable	209	RC-0291	...
1	salable	210	FR-R92B-58	...
1	salable	211	FR-R92R-58	...
...

- Instead of providing individual conditions, you can also use CASE to compare an expression to different values. Specify the expression between CASE and WHEN and for each WHEN for which you specify a value, the expression should be compared:

```
SELECT
    ProductLine as _ProductLine,
    CASE ProductLine
        WHEN 'R' THEN 'Road'
        WHEN 'M' THEN 'Mountain'
        WHEN 'T' THEN 'Touring'
        WHEN 'S' THEN 'Standard'
        ELSE 'other'
    END [Product Line]
,*
FROM dbo.DimProduct;
```

_ProductLine	Product Line	ProductKey	ProductAlternateKey	...
...
NULL	other	208	TP-0923	...
NULL	other	209	RC-0291	...
R	Road	210	FR-R92B-58	...
R	Road	211	FR-R92R-58	...
S	Standard	212	HL-U509-R	...

_ProductLine	Product Line	ProductKey	ProductAlternateKey	...
S	Standard	213	HL-U509-R	...
...

- Converting values is a classical use case for common table expressions (CTEs) in SQL, as well. In the following example, I specify common table expression `Class` as a unioned list of `SELECT` statements, which deliver a query result of three rows and two columns: one row per `Class` and an additional `ClassDescription`. I then join the table `DimProduct` with the common table expression (`Class`), as it would do with a table:

```
WITH Class AS (
  SELECT 'H' Class, 'High' ClassDescription UNION ALL
  SELECT 'M' Class, 'Medium' ClassDescription UNION ALL
  SELECT 'L' Class, 'Low' ClassDescription
)
SELECT
    dp.Class as _Class,
    c.ClassDescription as [Class Description]
    , *
FROM dbo.DimProduct dp
JOIN Class c ON c.Class=dp.Class;
```

_Class	Class	Class Description	ProductKey	ProductAlternateKey	...
...	M
Medium	194	SA-T612	...	L	Low
196	SD-2342	...	L	Low	198
SH-4562	...	H	High	210	FR-R92B-58
...

An improved version of this query would also take care of classes not listed in the CTE and show, e.g., “Unknown” as the `ClassDescription`. In the next example, I show you how to catch missing values and replace them with a meaningful text.

- I’m not a big fan of implementing business logic within a query (or report), as shown in the previous examples. In a perfect world, the report user would oversee defining what text should be shown instead of a flag or an indicator. It should not be necessary the query code to be updated (by IT) to implement a change. That’s why my preferred way of transforming flags and indicators is to create a physical table to store the lookup values. This table could sit inside an Excel workbook or a SharePoint list, but optimally, it’s a table in the database, exposed to the responsible users via an application, so they can maintain changes to the content of this table. This table is then simply joined in a query, as you can see from the following code example (which contains the code to re-create the table and fill it with initial values):

```

DROP TABLE IF EXISTS flag.Styles;
CREATE TABLE flag.Styles (
    Style char(1),
    StyleDescription nvarchar(50)
)
INSERT INTO flag.Styles
SELECT 'W' Class, 'Womens' StyleDescription UNION ALL
SELECT 'M' Class, 'Mens' StyleDescription UNION ALL
SELECT 'U' Class, 'Universal' StyleDescription
GO
CREATE OR ALTER VIEW flag.Style AS (
SELECT
    s.Style as _Style,
    ISNULL(s.StyleDescription, 'Unkown') as [Style Description]
    , *
FROM dbo.DimProduct dp
JOIN flag.Styles s ON s.Style=dp.Style
);

```

_Style	Style Description	ProductKey	ProductAlternateKey	...
...
NULL	Unkown	208	TP-0923	...
NULL	Unkown	209	RC-0291	...
U	Universal	210	FR-R92B-58	...
U	Universal	211	FR-R92R-58	...
...

- Unknown values are coded as NULL in relational databases. Power BI treats a NULL as zero for numerical columns, as *False* for True/False columns, and as empty for all other data types (text or Date/Time). It is best practice to avoid empty values in reports, though. Therefore, you need to replace NULL with meaningful text. If you want to write this as a condition in a CASE statement, remember that a comparison like <expression> = NULL never evaluates to true, but you need to formulate the condition as <expression> IS NULL. In the following example, I use the function ISNULL. This function checks if the first value IS NULL. If this is the case, then the second parameter is returned. If it is not the case, then the first parameter is returned. In cases where the logic is chained (“take the value from column c1, but if c1 is null, take c2, but if c2 is null take, ...”), you can use function COALESCE:

```

SELECT
    EnglishProductName,
    WeightUnitMeasureCode as WeightUnitMeasureCode,
    ISNULL(WeightUnitMeasureCode, 'N/A') [Weight Unit Measure Code]
FROM dbo.DimProduct dp

```


EnglishProductName	WeightUnitMeasureCode	Weight Unit Measure Code
...
Cone-Shaped Race	NULL	N/A
Reflector	NULL	N/A
LL Mountain Rim	G	G
ML Mountain Rim	G	G
...

So far, I have shown you how to split or combine existing tables or add calculations or meaningful text. Now it is time to talk about how you can create a new table from scratch.

Time and Date

If someone would present a data warehouse to me in which there is not a `Date` table, I would be very curious. Every single data warehouse I have built so far has had a `Date` table, and for good reasons. Creating such a table isn't a big issue. And keeping it automatically up-to-date (pun intended) can be done as well.⁵

To create the number of rows I need for the table, I use Itzik Ben-Gan's function `GetNumsItzikBatch`. It has two parameters (a start value and a maximum value) and will return a row for every value in between as a query result. I add then these numeric values via the function `DATEADD` as the number of days to my start date. The start date I derive as January 1 for the year of the earliest (MIN) order date of my fact table. The maximum value for the `GetNumsItzikBatch` function I derive as the difference in days between the start date and the last day of the year of the latest (MAX) order date of my fact table. All these components I created as common table expressions, as you can see in the first portion of the code listed here:

```
WITH
MinYear      AS
( SELECT YEAR(MIN(OrderDate)) MinYear FROM PowerBI.FactResellerSales ),
MinDate      AS
( SELECT DATEFROMPARTS(MinYear, 01, 01) MinDate FROM MinYear),
MaxYear      AS
( SELECT YEAR(MAX(OrderDate)) MaxYear FROM PowerBI.FactResellerSales),
MaxDate      AS
( SELECT DATEFROMPARTS(MaxYear+1, 12, 31) MaxDate FROM MaxYear),
MaxNumber    AS
( SELECT CONVERT(bigint, DATEDIFF(day, MinDate, MaxDate))
  MaxNumber FROM MinDate CROSS JOIN MaxDate),
```

⁵ The examples in this section use the *205 Time and Date.sql* file.

```

NumberTable AS
(
    SELECT N as Number
    FROM demo.GetNumsItzikBatch(0, (SELECT MaxNumber FROM MaxNumber))),
Date AS (
    SELECT
        DATEADD(
            day,
            n.Number,
            d.MinDate
        ) Date
    FROM
        NumberTable n
    CROSS JOIN MinDate d
)
SELECT
    Date
    CONVERT(int, FORMAT(Date, 'yyyyMMdd')) as DateKey,
    YEAR(Date) as Year,
    MONTH(Date) as [Month Number],
    FORMAT(Date, 'MMMM') as Month,
    FORMAT(Date, 'yyyy-MM') as [YYYY-MM],
    DATEPART(ww, Date) as [Week Number],
    DATEPART(iso_week, Date) as [Week Number ISO]
FROM
    Date;

```

The SELECT takes then the generated Date and applies different formulas to derive the columns I need in my Date dimension:

- DateKey is an integer representing the date in the format YYYYMMDD (e.g., 20231231).
- Year is the year (e.g., 2023).
- Month Number contains the number of the month (e.g., 12 for December).
- In Month usually stores the month's name (e.g., December).
- Depending on the needs of the report, I add a different format for the month (e.g., 2023-12). This is easy to do with the FORMAT function.
- Sometimes reports are based on weeks. Function DATEPART can deliver both the number of the week and the number of the ISO week.

The story for a Time table, if needed, is similar. This time, I generate a list of values between 1 and 1,440 (minutes a day). I find the code more readable when I specify 24 * 60, instead of 1,440, but that's, of course, totally up to you. I add these numbers, then, as minutes, to time "00:00:00," as you can see from the following code snippet:

```

WITH
NumberTable AS (
    SELECT N as Number FROM demo.GetNumsItzikBatch(1, 24
        /* hours */ * 60 /* minutes */),

```

```

Time AS (
SELECT
    DATEADD(minute, Number, '00:00:00') Time
FROM
    NumberTable
)
SELECT
    CONVERT(Time, Time)           as Time,
    FORMAT(Time, 'HH')           Hour,
    FORMAT(Time, 'mm')           Minutes,
    FORMAT(Time, 'HH:mm')        TimeDescription
FROM
    Time;

```

Again, I use function `FORMAT` to generate the additional columns for my `Time` table, based on what is needed in the reports. A `Time` table for every minute of a day contains only 1,440 rows—so we do not have to be greedy in terms of the columns the report users might ask you for.

Role-Playing Dimensions

In the world of SQL, role-playing dimensions can easily be solved, as you can join a table with a join predicate of your choice. The same tables could be joined with different join predicates multiple times in a query. One solution for Power BI and Analysis Services tabular is to add the same table multiple times under different names to the data model. As long as the dimension table is not too big, this is easily tolerable (and no, there is no clear number as to what “too big” means; it depends solely on the resources you are using).

Remember that best practice is to not directly access the tables from the data warehouse but to use `VIEWS` as a layer in between. So, the solution is to just create more than one `VIEW` for the identical table from the data warehouse. Instead of creating one view `Date`, you create variations per role: `[Order Date]`, `[Ship Date]`, etc. No need to actually duplicate the content of the dimension table in the data warehouse.

It’s important to remember that it is best practice to choose names for your columns that are unique through the whole data model. Nothing is more frustrating than discussions about “Which year are you showing in this visual?” or “Which date should the filter be applied on?”. That’s what the alias definition (keyword `AS`) in the `SELECT` projection is for:⁶

```

CREATE OR ALTER VIEW roleplaying.vw_OrderDate AS (
SELECT
    DateKey           as OrderDateKey,

```

⁶ The examples in this section use the *206 Role Playing Dimensions.sql* file.

```

        Date                as OrderDate,
        CalendarYear        as OrderYear
FROM
    PowerBI.DimDate
)
GO
CREATE OR ALTER VIEW roleplaying.vw_ShipDate AS (
SELECT
    DateKey                as ShipDateKey,
    Date                   as ShipDate,
    CalendarYear           as ShipYear
FROM
    PowerBI.DimDate
)

```

As long as the number of columns and the number of rows is small, creating these views and copy and pasting the prefix in front of all columns is doable. The brave ones among us will still choose a solution built on dynamic SQL, as it's really comfortable to use once you have built it. And, as for all automatic solutions, it will guarantee that the prefixes are spelled identically for all columns (which is not always the case when you make changes manually in an editor).

The following code first defines a bunch of variables to identify the database object the VIEW should be generated on (variables @SchemaName and @TableName), to specify the VIEW that should be generated (variables @SchemaNameTarget and @TableNameTarge), and to articulate the prefix that will be applied to the name of the VIEW and all columns. Another set of helper variables is defined to host the list of the names of the columns (@ColumnList) and the separator (@Separator).

Then, a SELECT statement aggregates, via function STRING_AGG, a string containing the column names of the source object in the format we need it in when creating the view (including brackets and the alias definition). For example, for a query list list containing only DateKey and Date, the aggregation creates this string: [DateKey] as [OrderDateKey], [Date] as [OrderDate]. This string is then printed to the console output with the rest of the text to form a valid CREATE OR ALTER VIEW statement:

```

-- Automate renaming
DECLARE
    @SchemaName sysname = 'PowerBI',
    @TableName sysname = 'DimDate',
    @SchemaNameTarget sysname = 'roleplaying',
    @TableNameTarget sysname = 'Date',
    @Prefix nvarchar(50) = 'Ship';

DECLARE
    @ColumnList nvarchar(max),
    @Separator nvarchar(50) = N',' + char(10) + char(09) -- + char(13)
;

```

```

SELECT
    @ColumnList = STRING_AGG(QUOTENAME(c.name) + ' AS ' +
        QUOTENAME(@Prefix + c.name), @Separator) WITHIN GROUP(
        ORDER BY c.column_id)
FROM
    (SELECT t.object_id, t.schema_id FROM sys.tables t UNION ALL
     SELECT v.object_id, v.schema_id FROM sys.views v) t
JOIN sys.columns c ON c.object_id=t.object_id
WHERE
    OBJECT_NAME(t.object_id) = @TableName AND
    SCHEMA_NAME(t.schema_id) = @SchemaName;
print '
CREATE OR ALTER VIEW ' + QUOTENAME(@SchemaNameTarget) + '.' +
    QUOTENAME('vw_' + @Prefix + @TableNameTarget) + ' AS (
SELECT
    ' + @ColumnList + '
FROM
    ' + QUOTENAME(@SchemaName) + '.' + QUOTENAME(@TableName) + '
)
'

```

To generate a view for a role-playing dimension, make sure to set the first five parameters right, execute the code, and then copy, paste, and execute the generated code from the console. Feel free to transform this code into a procedure, run the procedure in an automatic manner, etc.



I usually also add a comment in the code inside the generated VIEW's definition to warn people not to change this definition by hand, as their changes might be overwritten the next time I generate the code again. I, myself, was thankful for such a hint several times, when I, in the heat of a problem, was about to directly change the content of such a view.

Next, I introduce you how to implement slowly changing dimensions with the help of SQL.

Slowly Changing Dimensions

Implementing slowly changing dimensions (SCD) is only possible if you physically store the data in a data warehouse (a logical data warehouse layer with views on e.g., the data source does not allow for SCD). If you want (or need?) to implement slowly changing dimensions then there is no way around a data warehouse, where you persist the dimension's data. Only with the persisted data can you compare newly arriving data with the already existing data to track the changes. In this section I will show you how to implement slowly changing dimensions of Type 1 (last change wins) and Type 2 (creating separated rows for each version) in SQL, as well as explaining Type 0.

Example 18-2 and Table 18-1 illustrate the use cases:⁷

Example 18-2. Creating a source table and propagating it with sample rows

```
DROP TABLE IF EXISTS scd.SCDSOURCE;
CREATE TABLE scd.SCDSOURCE (
    AlternateKey int,
    Region nvarchar(50)
);
INSERT INTO scd.SCDSOURCE
SELECT 0, 'NA' UNION ALL
SELECT 1, 'Northwest' UNION ALL
SELECT 10, 'United Kingdom'
```

Table 18-1. Sample rows of source table

AlternateKey	Region
0	NA
1	Northwest
10	United Kingdom

Type 0: Retain Original

Type 0 means that the loaded data must not change. Implementation of the ETL task is limited to inserting new rows. The NOT EXISTS clause makes sure to identify such rows via the source's business key (column AlternateKey). When rows get deleted in the data source, nothing needs to be done in the data warehouse's table. The same applies to updated rows in the data source: the change is just be ignored. I store the creation date in an additional column (CreatedAt):

```
-- SCD Type 0
DROP TABLE IF EXISTS scd.SCD0;
CREATE TABLE scd.SCD0 (
    AlternateKey int,
    Region nvarchar(50),
    CreatedAt datetime2
);

-- INSERT
INSERT INTO scd.SCD0
SELECT AlternateKey, Region, SYSDATETIME()
FROM scd.SCDSOURCE stage
WHERE NOT EXISTS (SELECT TOP 1 1 FROM scd.SCD0 dwh WHERE dwh.AlternateKey =
    stage.AlternateKey)
```

⁷ The file used for the examples in this section is *207 Slowly Changing Dimensions.sql*.

AlternateKey	Region	CreatedAt
0	NA	2023-07-08 07:46:16.9373128
1	Northwest	2023-07-08 07:46:16.9373128
10	United Kingdom	2023-07-08 07:46:16.9373128

Type 1: Overwrite

I implement Type 1 in a way where it not only overwrites existing rows of data but keeps track of changes (e.g., timestamp of the change, which process changed the data, etc.). Checking changes for columns that can be NULL involves some extra logic to find out if the value changed to or from NULL. Instead of deleting rows, I only mark them as deleted (soft delete) by updating a column with the current timestamp. That's why rows that were deleted in the data source trigger an UPDATE (and not a DELETE) statement in the following code snippet.

In a first step, I load this table into the data warehouse (into another table). Let's make this table Type 1. This table contains all columns of the source table, plus some additional columns to store metadata: a timestamp, when the row was firstly created or the latest changes that happened, and a timestamp to mark it as deleted:

```
DROP TABLE IF EXISTS scd.SCD1
CREATE TABLE scd.SCD1 (
    AlternateKey int,
    Region nvarchar(50),
    ChangedAt datetime2,
    DeletedAt datetime2
);
```

A usual plain INSERT needs to be extended with a check to ensure that only rows that don't already exist are inserted into the data warehouse. The check for existence is done via the business key AlternateKey. ChangedAt is propagated with the current time and date, and DeletedAt is explicitly set to NULL ([Example 18-3](#)):

Example 18-3. Inserting rows into a table of type SCD 1

```
INSERT INTO scd.SCD1
SELECT AlternateKey, Region, SYSDATETIME(), null
FROM scd.SCDSource stage
WHERE NOT EXISTS (SELECT TOP 1 1 FROM scd.SCD1 dwh WHERE dwh.AlternateKey =
    stage.AlternateKey)
```

The result is that we've got the three rows in the data warehouse, as you can see here:

AlternateKey	Region	ChangedAt	DeletedAt
0	NA	2023-07-08 08:06:16.9373128	NULL
1	Northwest	2023-07-08 08:06:16.9373128	NULL
10	United Kingdom	2023-07-08 08:06:16.9373128	NULL

Let's introduce now some changes to the source table (laid out in [Example 18-4](#)). We cover the following cases:

- A new row appears in the source system (AlternateKey = 11). It must be inserted into the data warehouse. (That's already covered with the statement described in [Example 18-3](#). Make sure to rerun this statement.)
- A row could be removed from the source table (AlternateKey = 0). In this case, we expect the row's DeletedAt to be filled with the current timestamp.
- An attribute could be changed in the source (AlternateKey = 1, where I changed the region from "Northwest" to "Nordwest"). Then we expect the update to be reflected in the data warehouse and the ChangedAt to be set to the current point in time.
- A row could exist totally unchanged in the source system (AlternateKey = 10). We have to make sure to not update any column for such rows.

Example 18-4. Simulating changes to the data source

```
INSERT INTO scd.SCDSource SELECT 11, 'Austria'  
DELETE FROM scd.SCDSource WHERE AlternateKey=0;  
UPDATE scd.SCDSource SET Region='Nordwest' WHERE AlternateKey=1
```

In case of the removed row, I don't want to remove the row from the data warehouse, but only mark it as removed. That's why a deletion is implemented as an UPDATE that sets the DeletedAt column to the current timestamp:

```
UPDATE dwh  
SET  
    dwh.[DeletedAt] = SYSDATETIME()  
FROM [scd].[SCD1] dwh  
WHERE NOT EXISTS (SELECT TOP 1 1 FROM scd.SCDSource stage WHERE stage.  
    AlternateKey = dwh.AlternateKey)
```

An update is only justifiable if a change to one of the attributes of the row has happened. This is important; otherwise I would update the column ChangedAt under all circumstances (even when no change has happened). Someone reading this column would then wrongly get the impression that a change happened. And any delta-load

logic depending on the `ChangedAt` column would wrongly reload all rows instead, and I'd end up with an unwanted full load.

If an attribute is nullable, the check for a change is a bit more complex. It is not enough to just compare the columns from the source and from the data warehouse table with each other. If either would be `NULL`, the comparison would evaluate to `NULL` as well, and no update would happen. Instead, you need to additionally check if the source `IS NULL` and the target `IS NOT NULL` or if the source is `IS NOT NULL`, but the target `IS NULL`. In all these cases, we need to trigger the update as well. Also, we need to set `DeletedAt` to `NULL` to resurrect a row in case the row was previously (soft) deleted but happens to show up in the source again:

```
UPDATE [dwh]
SET
    [dwh].[Region] = [stage].[Region]
    , [dwh].[ChangedAt] = SYSDATETIME()
    , [dwh].[DeletedAt] = null
FROM [scd].[SCD1] [dwh]
INNER JOIN [scd].[SCDSOURCE] [stage] ON [dwh].AlternateKey=[stage].AlternateKey
WHERE
    ([dwh].[Region] <> [stage].[Region] OR
    ([dwh].[Region] IS NOT NULL AND [stage].[Region] IS NULL) OR
    ([dwh].[Region] IS NULL AND [stage].[Region] IS NOT NULL))
```



SQL Server 2022 (compatibility level 160+) introduced a new comparison operator `IS DISTINCT FROM`, which is available in all Azure tastes of SQL Server as well (Azure SQL Database, Azure SQL Managed Instance, SQL Endpoint in Microsoft Fabric, and Warehouse in Microsoft Fabric). With the help of this operator, you can replace the following:

```
([dwh].[Region] <> [stage].[Region] OR ([dwh].[Region]
IS NOT NULL AND [stage].[Region] IS NULL) OR ([dwh].
[Region] IS NULL AND [stage].[Region] IS NOT NULL))
```

with

```
[dwh].[Region] IS [NOT] DISTINCT FROM [stage].[Region]
```

This makes the code shorter and more readable.

If you've run all snippets as shown, the data warehouse table should look like this:

AlternateKey	Region	ChangedAt	DeletedAt
0	NA	2023-07-08 08:06:16.9373128	2023-07-08 08:39:58.6840573
1	Northwest	2023-07-08 08:06:16.9373128	NULL
10	United Kingdom	2023-07-08 08:06:16.9373128	NULL
11	Austria	2023-07-08 08:31:41.4968427	NULL

In a final step, I test if resurrection of previously deleted rows works as well. To practice this, please execute the script to initialize the source table (Example 18-2) and then rerun the load scripts (Examples 18-3, 18-5, and 18-6). This will reset the data warehouse to a state where the rows with DeletedAt of NULL matches the initial state, and ChangedAt has changed several times (except for the row with AlternateKey = 10, which never changed). A table of Type 1 will keep the extra row (AlternateKey=11) but marked as deleted. Compared to a table without slowly changing dimensions, the number of rows might be slightly bigger.

AlternateKey	Region	ChangedAt	DeletedAt
0	NA	2023-07-08 08:46:37.0275009	NULL
1	Northwest	2023-07-08 08:46:37.0275009	NULL
10	United Kingdom	2023-07-08 08:06:16.9373128	NULL
11	Austria	2023-07-08 08:31:41.4968427	2023-07-08 08:46:37.0275009

When you need to not only keep track that a change happened, but what a row looked like before the change, then you need to implement slowly changing dimensions of Type 2.

Type 2: Add New Row

A *slowly changing dimension Type 2* (SCD2) is similar to Type 1 for new and deleted rows but different for rows changed in the data source: in the case of a change of an attribute, you do not simply overwrite the row in the data warehouse but keep the old row (and mark it as old) and insert an additional row (with the changed attributes).

Before you can do the initial load, you need to create the table in the data warehouse. Instead of ChangedAt and DeletedAt, it contains ValidFrom and ValidUntil to specify a range of time when this version of a row was or is active. It's important to point out that a surrogate key (SID) is mandatory in this scenario. The business key (AlternateKey) will not be unique, as there will (soon) be multiple versions of this key stored in the table:

```
DROP TABLE IF EXISTS scd.SCD2
CREATE TABLE scd.SCD2 (
    SID int identity(1,1),
    AlternateKey int,
    Region nvarchar(50),
    ValidFrom datetime2,
    ValidUntil datetime2
);
```

Inserting into a Type 2 table is similar to inserting into a Type 1 table (see Example 18-5). Ensure that an active row for this business key (AlternateKey) doesn't already exist. A row is active if ValidUntil is NULL. ValidFrom is propagated

with the current timestamp, and ValidUntil is explicitly set to NULL for this new row. (Alternatively, you could also set it to a date in the far future, like December 31 in year 9999. This has, though, the potential to create huge date tables in Power BI in case users don't follow best practices and either do not turn off the auto date/time setting, as described in [“Time and Date” on page 116](#), or use function CALENDARAUTO, as described in [“Time and Date” on page 203](#)).

Example 18-5. Inserting rows into a table of type SCD2

```

INSERT INTO [scd].[SCD2] ([AlternateKey], [Region], [ValidFrom], [ValidUntil])
SELECT [stage].[AlternateKey], [stage].[Region], SYSDATETIME() AS [ValidFrom],
       null AS [ValidUntil]
FROM [scd].[SCDSource] [stage]
WHERE NOT EXISTS (
    SELECT TOP 1 1
    FROM [scd].[SCD2] [dwh]
    WHERE [dwh].AlternateKey=[stage].AlternateKey
          AND [dwh].[ValidUntil] IS NULL
)

```

The result is that you've got the three rows in the data warehouse, as you can see in the following table:

SID	AlternateKey	Region	ValidFrom	ValidUntil
1	0	NA	2023-07-08 10:00:06.9957244	NULL
2	1	Northwest	2023-07-08 10:00:06.9957244	NULL
3	10	United Kingdom	2023-07-08 10:00:06.9957244	NULL

Let's now introduce the same changes to the source table as in the Type 1 scenario (laid out in [Example 18-4](#)). After applying the changes to the source table, make sure to rerun [Example 18-5](#).

In case of the removed row, I don't want to remove the row from the data warehouse but only mark it as removed. That's why a delete is implemented as an UPDATE statement that sets the ValidUntil column to the current timestamp ([Example 18-6](#)).

Example 18-6. Soft deleting rows in a table of type SCD2

```

UPDATE dwh
SET dwh.[ValidUntil] = SYSDATETIME()
FROM [scd].[SCD2] dwh
WHERE
    dwh.SID >= 1 AND
    ISNULL(dwh.[ValidUntil], SYSDATETIME()) >= SYSDATETIME() AND
    NOT EXISTS (SELECT TOP 1 1 FROM [scd].[SCDSource] stage
                WHERE dwh.AlternateKey=stage.AlternateKey)

```

An update is only justifiable if a change to one of the attributes of the row has happened. This is important, as otherwise we would update the column ValidUntil for all existing rows and re-create new versions for the AlternateKey. The dimension table would unnecessarily insert a row for every AlternateKey with every single load. In the case of a daily load, after a year, 365 versions of every region would exist, even when no column changes its values. Therefore, make sure to test this case thoroughly.

Nullable attributes need a special treatment, as well (also described in the section about Type 1). An update must be done in two steps: soft delete the old version via UPDATE, then insert the new version of the row, as you can see in [Example 18-7](#).

Example 18-7. Updating a row in a table of type SCD2

```
-- INACTIVATE OLD VERSION
UPDATE dwh
SET dwh.[ValidUntil] = SYSDATETIME()
FROM [scd].[SCD2] dwh
INNER JOIN [scd].[SCDSource] stage on dwh.AlternateKey=stage.AlternateKey
WHERE
    dwh.SID >= 1 AND
    ISNULL(dwh.[ValidUntil], SYSDATETIME()) >= SYSDATETIME() AND
    ([dwh].[Region] <> [stage].[Region] OR ([dwh].[Region]
        IS NOT NULL AND [stage].[Region] IS NULL) OR ([dwh].[Region]
        IS NULL AND [stage].[Region] IS NOT NULL))
-- INSERT NEW VERSION
INSERT INTO [scd].[SCD2] ([AlternateKey], [Region], [ValidFrom], [ValidUntil])
SELECT [stage].[AlternateKey], [stage].[Region], SYSDATETIME() AS [ValidFrom],
    null AS [ValidUntil]
FROM [scd].[SCDSource] stage
WHERE
    EXISTS (SELECT TOP 1 1 FROM [scd].[SCD2] dwh WHERE dwh.AlternateKey=
        stage.AlternateKey
    AND ([dwh].[Region] <> [stage].[Region] OR ([dwh].[Region]
        IS NOT NULL AND [stage].[Region] IS NULL) OR ([dwh].[Region]
        IS NULL AND [stage].[Region] IS NOT NULL))
    AND dwh.SID >= 1
    )
```

If you have executed all snippets as described, the data warehouse table will now look like the following:

SID	AlternateKey	Region	ValidFrom	ValidUntil
1	0	NA	2023-07-08 10:00:06.9957244	2023-07-08 10:09:27.5110675
2	1	Northwest	2023-07-08 10:00:06.9957244	2023-07-08 10:14:11.3859165
3	10	United Kingdom	2023-07-08 10:00:06.9957244	NULL
4	11	Austria	2023-07-08 10:04:56.1362259	NULL
5	1	Nordwest	2023-07-08 10:14:11.3859165	NULL

In a final step, I want to test if resurrection of previously deleted rows works as well. To practice this, please execute the script to initialize the source table ([Example 18-2](#)) and then rerun the load scripts ([Examples 18-7](#), [18-8](#), and [18-9](#)). This will reset the data warehouse, to a state where the active rows match the initial state. You can clearly see that a table with Slowly Changing Dimensions Type 2 can grow very fast in, number of rows, as every change (and re-change) creates its own row:

SID	AlternateKey	Region	ValidFrom	ValidUntil
1	0	NA	2023-07-08 10:00:06.9957244	2023-07-08 10:09:27.5110675
2	1	Northwest	2023-07-08 10:00:06.9957244	2023-07-08 10:14:11.3859165
3	10	United Kingdom	2023-07-08 10:00:06.9957244	NULL
4	11	Austria	2023-07-08 10:04:56.1362259	2023-07-08 10:18:46.0263334
5	1	Nordwest	2023-07-08 10:14:11.3859165	2023-07-08 10:18:46.0263334
6	0	NA	2023-07-08 10:18:46.0263334	NULL
7	1	Northwest	2023-07-08 10:18:46.0263334	NULL

Another challenge that can happen with every dimension is that it can store hierarchical information. Continue to read if you want to learn how you can use SQL to transform this information into a shape that can be consumed easily by Power BI.

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 [“Denormalizing” on page 375](#). With the natural hierarchy denormalized, you have all levels of the hierarchy as columns in one table. Adding them to a hierarchy is easy.

Here I want to concentrate on parent-child hierarchies. They are very common, and for the sake of Power BI, you need to store the names of all parent levels in dedicated columns. Therefore, you need to bring the parent-child hierarchy into a materialized path.

Before you can create the materialized path for the hierarchy, you have to merge and expand the query as many times as levels you have. You could do this by writing a JOIN per (expected) level of the parent-child hierarchy, but it would have the drawback of duplicated code (similar to the first, static, solution in Power Query, described in [“Hierarchies” on page 286](#)). While in Power Query, you need a function to solve this problem in a dynamic way. You can use a common table expression to solve this within a query. The result delivers, then, one row per level. To turn those

rows into columns, you need to PIVOT the result. The query in [Example 18-8](#) shows the full code.⁸

Example 18-8. Pivoting a self-referencing hierarchy.

```
;WITH PCCTE AS (  
  SELECT  
    EmployeeKey, ParentEmployeeKey,  
    convert(varchar(max), FirstName + ' ' + LastName) as FullName,  
    1 as Lvl,  
    convert(varchar(max), FirstName + ' ' + LastName) as [Path],  
    CONVERT(bit, CASE WHEN EXISTS (SELECT 1 FROM dbo.DimEmployee sde  
      WHERE sde.ParentEmployeeKey = DimEmployee.EmployeeKey)  
      THEN 0 ELSE 1 END) IsLeaf  
  FROM dbo.DimEmployee  
  WHERE ParentEmployeeKey IS NULL  
  UNION ALL  
  SELECT  
    child.EmployeeKey, child.ParentEmployeeKey,  
    convert(varchar(max), FirstName + ' ' + LastName) as FullName,  
    parent.Lvl + 1 as Lvl,  
    convert(varchar(max), parent.[Path] + '|' + child.FirstName + ' ' + child.  
      LastName) as [Path],  
    CONVERT(bit, CASE WHEN EXISTS (SELECT 1 FROM dbo.DimEmployee sde  
      WHERE sde.ParentEmployeeKey = child.EmployeeKey)  
      THEN 0 ELSE 1 END) IsLeaf  
  FROM dbo.DimEmployee child  
  JOIN PCCTE parent ON parent.EmployeeKey= child.ParentEmployeeKey  
)  
SELECT *  
FROM (  
  SELECT c.*, [split].[Value], 'Level ' + convert(varchar, ROW_NUMBER() OVER(  
    PARTITION BY EmployeeKey ORDER BY [Lvl] DESC)) AS [ColumnName]  
  FROM PCCTE AS c  
  CROSS APPLY STRING_SPLIT([Path], '|') AS split  
) AS t  
PIVOT(  
  MAX([Value])  
  FOR ColumnName  
  IN([Level 1], [Level 2], [Level 3], [Level 4], [Level 5], [Level 6],  
    [Level 7])  
) p
```

⁸ The examples in this section use the *208 Hierarchies.sql* file.



CTE can divide the code of a query into smaller parts, which are easier for the developer and the database management system to digest. I sometimes call CTEs “named subqueries”: you define a query, name it, and then join it in the main portion of your `SELECT` statement. But there’s more: you can refer to a CTE within its own definition. This will start iterations over the resulting content of the CTE. Such a (recursive) CTE must start with a query that doesn’t reference itself. This is called the *anchor query* of the recursive CTE. Then you `UNION` this `SELECT` with a second `SELECT` statement, which references the CTE. When the CTE is referenced during the first iteration, its content is simply what the first (*anchor*) query returns. During the second iteration, the CTE contains what the first iteration contained. And so forth until an iteration doesn’t return any rows, or until a maximum iteration is reached (which can be set via options, including 0 for no limit).

Don’t forget to add some logic to your measures, as described in “Hierarchies” on page 210, to avoid this hierarchy being expanded to unnecessary levels.

Key Takeaways

In this chapter, I showed you how you can apply the power of SQL to bring a data model into a star schema and solve typical use cases. Specifically, you learned the following:

- Normalizing your fact tables involves steps to find candidates for dimensions. `GROUP BY` and `COUNT(*)` are helpful to get an idea of the cardinality of a column, even before you load it into the data model.
- You create a (denormalized) dimension table by joining all necessary tables and creating a `DISTINCT` list of columns.
- Don’t spend too much time on calculations in SQL. DAX is usually the better place.
- Physically adding variations for a table (for role-playing purposes) is very easy: you just create several views with renamed columns, based on the same query.
- You can flatten parent-child-hierarchies by applying several steps. You learned about recursive *common table expressions* and the `PIVOT` keyword in SQL.

Continue with the next chapter to learn how to support advanced data modeling use cases in SQL.

Real-World Examples Using SQL

Chapter 3 introduces real-world use cases I've encountered while working on projects for customers and how I solved them in Power BI. Being able to “attack” a problem with different “weapons” is of help when it comes to those challenges. You always need to be flexible and sometimes think out-of-the box to find the right approach. In this chapter, I demonstrate how to generate and shape tables in SQL:

- You'll learn how to generate both the bin table and the bin range table in SQL.
- To resolve a many-to-many relationship (like for a budget that is on a different level of granularity than the actual values), you need to create a *bridge* table. This can be done easily in SQL.
- For the demonstrated solution for multi-language reports, you need a table containing the texts for headlines, buttons, etc. I will show you how you can pivot the table, so it perfectly fits the solution.
- Key-value pair tables are hard to query in the shape they are in natively. Therefore, you will learn how you can pivot the table to bring it in an analytics-friendly shape.
- I'll show you what role a data warehouse plays in the concept of self-service BI versus enterprise BI.

Binning

In “Binning” on page 58, I describe three solutions. All of the necessary tables can be created in SQL. Remember that I wouldn't recommend adding the bin information into the fact table, as a change in the bin ranges would mean updating the whole fact

table. That’s why you should look at the two other solutions only. The file used in this section’s examples is *301 Binning.sql*.

Deriving the Lookup Table from the Facts

Creating a DISTINCT list of values for the lookup table (which contains, then, every possible value and its bin) in SQL is easy, as there is a dedicated keyword for that. The name of the bin can be created with the help of the CASE keyword. CASE is similar to DAX’s SWITCH function, which I introduce in “Binning” on page 216:

```
CREATE OR ALTER VIEW bin.vw_QuantityBin AS
SELECT
    DISTINCT
    OrderQuantity,
    CASE
    WHEN OrderQuantity <= 5 THEN 'Small'
    WHEN OrderQuantity <= 10 THEN 'Medium'
    ELSE 'Large'
    END QuantityBin
FROM
    PowerBI.FactResellerSales
```

OrderQuantity	QuantityBin
33	Large
1	Small
6	Medium
38	Large
12	Large
...	...

To use this bin table in a query, you join view vw_QuantityBin with the fact table on the OrderQuantity column. This looks a bit unusual, as the OrderQuantity column in the fact table is not a classic foreign key, but we use it in such a way. This frees you from the burden of needing to create an additional foreign key in the fact table (which would occupy space and would need to be maintained if the ranges of the bins should change):

```
SELECT
    frs.OrderQuantity,
    qb.QuantityBin,
    frs.*
FROM
    PowerBI.FactResellerSales frs
JOIN bin.vw_QuantityBin qb ON
    qb.OrderQuantity = frs.OrderQuantity
```

Due to a high number of distinct values, the bin table could be huge. This is the case for the SalesAmount column. To reduce the number of rows, you could create this bin list not on the fact table's value, but, e.g., on the thousandths of the value. You simply take the preceding code snippet and replace OrderQuantity with SalesAmount /1000, which can be aliased as SalesAmountK. The changed code is listed here:

```
CREATE OR ALTER VIEW bin.vw_SalesAmountBin AS
SELECT
    DISTINCT
    CONVERT(int, SalesAmount/1000) SalesAmountK,
    CASE
    WHEN SalesAmount/1000 <= 5 THEN 'Small'
    WHEN SalesAmount/1000 <= 10 THEN 'Medium'
    ELSE 'Large'
    END SalesAmountBin
FROM
    PowerBI.FactResellerSales
```

SalesAmountK	SalesAmountBin
1	Small
6	Medium
12	Large
26	Large
...	...

As the goal is, again, to not add columns to the fact table you need to implement the division by 1,000 in the JOIN predicate:

```
SELECT
    frs.SalesAmount,
    sb.SalesAmountBin,
    frs.*
FROM
    PowerBI.FactResellerSales frs
JOIN bin.vw_SalesAmountBin sb ON
    sb.SalesAmountK = CONVERT(int, frs.SalesAmount/1000)
```

Generating a Lookup Table

Depending on the size of the fact table, querying the distinct values of the fact table to create the bin table can result in a bad performance and put pressure on your resources, leading to a long query duration. The following code queries the fact table too, but asks for the smallest and largest OrderQuantity, which can sometimes be calculated more efficiently. That's why I included this version here as well:

```
CREATE OR ALTER VIEW bin.vw_QuantityBin AS
WITH
    MinNumber AS (SELECT MIN(OrderQuantity) MinNumber FROM PowerBI.
```

```

FactResellerSales),
MaxNumber AS (SELECT MAX(OrderQuantity) MaxNumber FROM PowerBI.
FactResellerSales),
NumberTable AS (
SELECT N as Number
FROM demo.GetNumsItzikBatch(0, (SELECT MaxNumber FROM MaxNumber)))
SELECT
    Number as OrderQuantity,
    CASE
    WHEN Number <= 5 THEN 'Small'
    WHEN Number <= 10 THEN 'Medium'
    ELSE 'Large'
    END QuantityBin
FROM
    NumberTable

```

OrderQuantity	QuantityBin
0	Small
1	Small
2	Small
3	Small
4	Small
5	Small
6	Medium
7	Medium
...	...

Range Table

A completely different way of looking at the problem is not to create a table containing all possible values and their bins but a table that specifies the ranges per bin, with a lower and an upper value. You can create such a lookup table if you UNION a simple SELECT statement that provides the bin's name and the ranges. But I guess it would be better if a domain user, rather than a SQL developer, has sole control over the names and ranges. Here's the code:

```

CREATE OR ALTER VIEW bin.[vw_QuantityBin Range] AS (
SELECT 'Small' [QuantityBin], null [Low (incl.)], 5 [High (excl.)] UNION ALL
SELECT 'Medium' [SalesAmountBin], 5 [Low (incl.)], 10 [High (excl.)] UNION ALL
SELECT 'Large' [QuantityBin], 10 [Low (incl.)], null [High (excl.)]
)

```

QuantityBin	Low (incl.)	High (excl.)
Small	NULL	5
Medium	5	10
Large	10	NULL

The created table is much smaller, but using the table in a join is slightly more complex: you need to get the non-equi-joins (\geq and $<$) into the correct shape and treat the NULLs in the ranges correctly:

```
SELECT
    frs.OrderQuantity,
    sb.QuantityBin,
    frs.*
FROM
    PowerBI.FactResellerSales frs
JOIN bin.[vw_QuantityBin Range] sb ON
    (frs.OrderQuantity >= sb.[Low (incl.)] OR sb.[Low (incl.)] IS NULL)
    AND
    (frs.OrderQuantity < sb.[High (excl.)] OR sb.[High (excl.)] IS NULL)
```

OrderQuantity	QuantityBin	OrderDateKey	...
...
3	Small	20231101	...
3	Small	20231101	...
6	Medium	20201201	...
5	Medium	20201201	...
...

Next, I show you how to create the bridge table discussed in “Budget” on page 60 as a solution to avoid many-to-many relationships.

Budget

The “long story short” in “Budget” on page 60 is that you need a bridge table between the Budget table and the Product table because the Product Group column isn’t unique in either of the two tables. Power BI will only allow creating a relationship of a many-to-many cardinality based on Product Group. Such relationships have more unintended effects in Power BI than one-to-many relationships. The bridge table allows you to transform the many-to-many relationship into two one-to-many relationships.

Creating the bridge table is a very simple SELECT statement with DISTINCT over the Product Group column of each table. Then you combine these two queries with a UNION:¹

```
CREATE OR ALTER VIEW budget.ProductGroup AS
SELECT DISTINCT ProductGroup from budget.Product
UNION
SELECT DISTINCT ProductGroup from budget.Budget
```

ProductGroup

Group 1

Group 2

Group 3



When creating a bridge table, you need to write a UNION and not a UNION ALL. UNION will scan the result for duplicates and remove them. UNION ALL would keep Product Group values that appear in both the Product table and the Budget table (e.g., “Group 2”). The bridge table must not contain duplicates.

The next section cover how to implement pivoting with SQL.

Multi-Language Model

For this section’s examples, I use the *303 Localized model.sql* file and concentrate on the Textcomponent table that contains the parts of a report (headlines, buttons, etc.) which are usually static. Such a table for two pieces of text (Sales Overview and Sales Details) and languages—English (EN) and Klingon (tlh-Latn)—could look like the following:

```
CREATE OR ALTER VIEW [language].[TextComponent] AS
(
SELECT 'EN' [LanguageID], 'SalesOverview' [TextComponent],
'Sales Overview' [DisplayText]
UNION ALL
SELECT 'EN' [LanguageID], 'SalesDetails' [TextComponent],
'Sales Details' [DisplayText]
UNION ALL
SELECT 'tlh-Latn' [LanguageID], 'SalesOverview' [TextComponent],
'QI'yaH' [DisplayText]
UNION ALL
SELECT 'tlh-Latn' [LanguageID], 'SalesDetails' [TextComponent],
```

¹ The examples in this section use the *302 Budget.sql* file.

```
'qeylIS belHa'' [DisplayText]
)
```

LanguageID	TextComponent	DisplayText
EN	SalesOverview	Sales Overview
EN	SalesDetails	Sales Details
tlh-Latn	SalesOverview	Ql'yaH
tlh-Latn	SalesDetails	qeyllS belHa'

This table can easily be used after it's been pivoted (one column per piece of text, one row per language). As it will be in an active one-to-many relationship with the Language table, only one row (one language) will be available at query time.

Fortunately, SQL provides the PIVOT keyword to transform rows into columns. You need to provide the following:

- An aggregate function and the name of the column whose content you want pivot into different columns. For text columns, you choose either the MIN or the MAX function, which will return the alphabetically first or last text, in case there is more than one row available for a piece of text. Even if you are convinced that this is not the case, using PIVOT is mandatory to provide such an aggregate function.
- The name of the column whose content should be transformed into column names, after the FOR keyword.
- A list of the new column names (a list of the expected content of the column name whose content should be transformed into column names) provided after the IN keyword.

The code looks straightforward:

```
CREATE OR ALTER VIEW [language].[vw_TextComponent]
SELECT
    *
FROM
    [language].[TextComponent] tc
PIVOT
(
    MIN([DisplayText])
    FOR [TextComponent]
    IN (
        [SalesOverview],
        [SalesDetails]
    )
) p
```

LanguageID	SalesOverview	SalesDetails
EN	Sales Overview	Sales Details
tlh-Latn	Ql'yaH	qeyllIS belHa'

On top of my wish list for the SQL syntax is to be able to provide the list of column names for the IN keyword as a subquery. Unfortunately, this is not supported by the language. Therefore, you need to either manually update the code every single time a new value for TextComponent is added to the table, or reach out to dynamic SQL to generate the whole SELECT statement including the list of column names, as you can see here:

```

DECLARE
    @CRLF nvarchar(MAX) = CHAR(13)+CHAR(10),
    @cmd nvarchar(MAX),
    @ColumnNameList nvarchar(max)

SELECT
    @ColumnNameList = STRING_AGG([ColumnNameKey], ', ')
    WITHIN GROUP (ORDER BY [ColumnNameKey])

FROM
    (SELECT
        DISTINCT
        CONVERT(nvarchar(max), [TextComponent] + @CRLF)
        as [ColumnNameKey]

    FROM
        (
            SELECT
                DISTINCT
                QUOTENAME(TRIM(tc.[TextComponent])) [TextComponent]
            FROM [language].[TextComponent] tc
        ) k
    ) x;

-- VIEW
SET @cmd=N'
CREATE OR ALTER VIEW [language].[vw_TextComponent]
AS
(
    /** DO NOT MAKE ANY CHANGES DIRECTLY ***/
    /** This code was generated ***/
    SELECT *
    FROM [language].[TextComponent] tc
    PIVOT
    (
        MIN([DisplayText])
        FOR [TextComponent] IN (
            + @ColumnNameList + N'
        )
    ) as p
'

exec sp_executesql @stmt = @cmd

```




When using dynamic SQL, always ensure that SQL injection is prohibited. You can find a good guide for this at [Microsoft](#).

In this book's code examples, the problem of SQL injection is rather negligible, as the injected text is not an input from a (possibly untrustworthy) user but from a key-value pair table, delivered from an application and from a lookup table maintained by you. On top of that, the function QUOTENAME will wrap every key inside of brackets. In case the name of a key contains harmful code, it would simply not be executed but treated as part of the key's name only.

I first declared the variables. One contains the two characters for carriage return and line feed (@CRLF), so that list of column names can be put one per line. Variable @cmd hosts the whole SQL statement to create the view. And @ColumnNameKey will contain the list of column names, which is used for the IN clause for PIVOT.

The @ColumnNameList is propagated in the inner SELECT with DISTINCT values of the TextComponent column. The values are trimmed (function TRIM) to cut off leading and trailing whitespace. Function QUOTENAME wraps the content inside brackets ([]) and will add escape characters in case the TextComponent contains any "]" character itself. I then append @CRLF and convert the content explicitly to a varchar(max) so no column name is cut off. In the outermost SELECT, I aggregate all the column names with function STRING_AGG and a comma (",") as a separator in alphabetical order.

The @cmd variable is then assigned to a string containing the full CREATE OR ALTER VIEW command, where I replace a hardcoded list for IN with the content of @ColumnNameList. Here is the generated command:

```
CREATE OR ALTER VIEW [language].[vw_TextComponent]
AS
(
  --/*** DO NOT MAKE ANY CHANGES DIRECTLY ***/
  --/*** This code was generated ***/
  SELECT      *
  FROM [language].[TextComponent] tc
  PIVOT
  (
    MIN([DisplayText])
    FOR [TextComponent] IN (
      [SalesDetails]
      , [SalesOverview]
    )
  ) as p
```

In the next section, you’ll face a similar challenge: pivoting a table. Additionally, it will be necessary to find the correct data types for the columns, which adds a bit of complexity to the problem.

Key-Value Pair Tables

The problem for the key-value pair table is similar to that of the TextComponent table—it needs to be pivoted. There is an additional challenge, though. While all columns in the pivoted TextComponent table are of a string data type (varchar), the types for the columns of the pivoted key-value pair table can be of any data type.²

Here’s the table:

```
CREATE TABLE [dwh].[KeyValue] (  
    [_Source] varchar(20),  
    [ID] int,  
    [Key] nvarchar(3000),  
    [Value] nvarchar(3000),  
    [Type] nvarchar(125)  
)  
;  
  
INSERT INTO [dwh].[KeyValue] VALUES  
( '[dwh].[KeyValue]', 1, 'name', 'Bill', 'text'),  
( '[dwh].[KeyValue]', 1, 'city', 'Seattle', 'text'),  
( '[dwh].[KeyValue]', 2, 'name', 'Jeff', 'text'),  
( '[dwh].[KeyValue]', 2, 'city', 'Seattle', 'text'),  
( '[dwh].[KeyValue]', 3, 'name', 'Markus', 'text'),  
( '[dwh].[KeyValue]', 3, 'city', 'Alkoven', 'text'),  
( '[dwh].[KeyValue]', 1, 'revenue', '20000', 'Int64.Type'),  
( '[dwh].[KeyValue]', 2, 'revenue', '19000', 'Int64.Type'),  
( '[dwh].[KeyValue]', 3, 'revenue', '5', 'Int64.Type'),  
( '[dwh].[KeyValue]', 1, 'firstPurchase', '1980-01-01', 'date'),  
( '[dwh].[KeyValue]', 2, 'firstPurchase', '2000-01-01', 'date'),  
( '[dwh].[KeyValue]', 3, 'firstPurchase', '2021-01-01', 'date'),  
( '[dwh].[KeyValue]', 1, 'zip', '0100', 'text'),  
( '[dwh].[KeyValue]', 2, 'zip', '0200', 'text'),  
( '[dwh].[KeyValue]', 3, 'zip', '0300', 'text')
```

_Source	ID	Key	Value	Type
[dwh].[KeyValue]	1	name	Bill	text
[dwh].[KeyValue]	1	city	Seattle	text
[dwh].[KeyValue]	2	name	Jeff	text
[dwh].[KeyValue]	2	city	Seattle	text

2 The examples in this section use the *304 Key Value.sql* file.

_Source	ID	Key	Value	Type
[dwh].[KeyValue]	3	name	Markus	text
[dwh].[KeyValue]	3	city	Alkoven	text
[dwh].[KeyValue]	1	revenue	20,000	Int64.Type
[dwh].[KeyValue]	2	revenue	19,000	Int64.Type
[dwh].[KeyValue]	3	revenue	5	Int64.Type
[dwh].[KeyValue]	1	firstPurchase	1980-01-01	date
[dwh].[KeyValue]	2	firstPurchase	2000-01-01	date
[dwh].[KeyValue]	3	firstPurchase	2021-01-01	date
[dwh].[KeyValue]	1	zip	0100	text
[dwh].[KeyValue]	2	zip	0200	text
[dwh].[KeyValue]	3	zip	0300	text

I will first teach you how to write the static code before jumping to dynamic SQL. When you apply the PIVOT keyword in the same way as in “Multi-Language Model” on page 404, you’ll be disappointed because you’ll get more rows than expected, with many column values being NULL:

```
-- PIVOT
SELECT
    *
FROM
    [dwh].[KeyValue] kv
PIVOT
(
    MIN([Value])
    FOR [Key]
    IN (
        [name],
        [city],
        [revenue],
        [firstPurchase],
        [zip]
    )
) p
```

_Source	ID	Type	name	city	revenue	firstPurchase	zip
[dwh].[KeyValue]	1	date	NULL	NULL	NULL	1980-01-01	NULL
[dwh].[KeyValue]	1	Int64.Type	NULL	NULL	20,000	NULL	NULL
[dwh].[KeyValue]	1	text	Bill	Seattle	NULL	NULL	0100
[dwh].[KeyValue]	2	date	NULL	NULL	NULL	2000-01-01	NULL
[dwh].[KeyValue]	2	Int64.Type	NULL	NULL	19,000	NULL	NULL
[dwh].[KeyValue]	2	text	Jeff	Seattle	NULL	NULL	0200
[dwh].[KeyValue]	3	date	NULL	NULL	NULL	2021-01-01	NULL

_Source	ID	Type	name	city	revenue	firstPurchase	zip
[dwh].[KeyValue]	3	Int64.Type	NULL	NULL	5	NULL	NULL
[dwh].[KeyValue]	3	text	Markus	Alkoven	NULL	NULL	0300

You could GROUP BY column ID and apply function MIN on all pivoted columns. Before you do that, let's think about why you receive these extra rows. The reason for this behavior can be found in the Type column. There are several Types per ID. Removing the column Type from the query will also remove the extra rows. You need to tell SQL to ignore this column for the sake of pivoting. That's why I rewrote the query with a subselect in the first FROM clause, replacing the simple reference to the table. The subselect queries only the necessary columns (ID, Key, and Value):

```
-- PIVOT without Type column
SELECT
    *
FROM
    (SELECT [ID], [Key], [Value] FROM [dwh].[KeyValue]) kv
PIVOT
(
    MIN([Value])
    FOR [Key]
    IN (
        [name],
        [city],
        [revenue],
        [firstPurchase],
        [zip]
    )
) p
```

ID	name	city	revenue	firstPurchase	zip
1	Bill	Seattle	20,000	1980-01-01	0100
2	Jeff	Seattle	19,000	2000-01-01	0200
3	Markus	Alkoven	5	2021-01-01	0300

The next improvement is to get the data types right. PIVOT will keep the data type of the Value column and will make all columns in the result set a VARCHAR(3000). Because there's simply no guarantee that the Value for a Key really sticks to the expected data type (described in the Type column), I prefer to use the function TRY_CONVERT. In case a value cannot be converted to the desired data type, this function does not throw an error (as function CONVERT would do) but simply returns NULL as the result after conversion. You need to decide on your own if you prefer an error or NULL.

Here is the improved code for the VIEW:

```

CREATE OR ALTER VIEW [PowerBI].[KeyValue]
AS
(
SELECT
    p.ID [ID]
    , TRY_CONVERT(NVARCHAR(3000), [city]) AS [city]
    , TRY_CONVERT(DATE, [firstPurchase]) AS [firstPurchase]
    , TRY_CONVERT(NVARCHAR(3000), [name]) AS [name]
    , TRY_CONVERT(BIGINT, [revenue]) AS [revenue]
    , TRY_CONVERT(NVARCHAR(3000), [zip]) AS [zip]
FROM (SELECT [ID], [Key], [Value] FROM [dwh].[KeyValue]) kv
PIVOT
    (MIN([Value]) FOR [Key] IN (
        [city]
        , [firstPurchase]
        , [name]
        , [revenue]
        , [zip]
    )
) as p
)

```

I bet you already see the drawback of this static code: if new keys are added, you always need to adopt the definition of the VIEW. You need to maintain the column list in the projection (SELECT) and in the PIVOT list. You also need to choose the right data type per column. If, over time, many new keys are added, maintaining the definition of the VIEW can take up a lot of your time. That's where dynamic SQL can come in. First, create a table that matches the content of the Type column to a data type available in SQL:

```

-- KeyType
DROP TABLE IF EXISTS [KeyValue].[KeyType];
CREATE TABLE [KeyValue].[KeyType] (
    KeyType nvarchar(128),
    KeyDescription nvarchar(128),
    DataType nvarchar(128)
)
INSERT INTO [KeyValue].[KeyType] VALUES
(N'text', N'Text', N'NVARCHAR(3000)' ),
(N'Int64.Type', N'Whole Number', N'BIGINT'),
(N'number', N'Decimal Number', N'DOUBLE' ),
(N'currency', N'Fixed Decimal Number', N'DECIMAL(19,4)' ),
(N'Percentage', N'Percentage', N'DECIMAL(19,4)' ),
(N'datetime', N'Date/Time', N'DATETIME2' ),
(N'date', N'Date', N'DATE' ),
(N'time', N'Time', N'TIME' ),
(N'datetimezone', N'Date/Time/Timezone', N'DATETIMEOFFSET' ),
(N'duration', N'Duration', N'DOUBLE' ),
(N'logical', N'True/False', N'BIT' ),
(N'binary', N'Binary', N'VARCHAR(max)' )

```

KeyType	KeyDescription	DataType
text	Text	NVARCHAR(3000)
Int64.Type	Whole Number	BIGINT
number	Decimal Number	DOUBLE
currency	Fixed Decimal Number	DECIMAL(19,4)
Percentage	Percentage	DECIMAL(19,4)
datetime	Date/Time	DATETIME2
date	Date	DATE
time	Time	TIME
datetimezone	Date/Time/Timezone	DATETIMEOFFSET
duration	Duration	DOUBLE
logical	True/False	BIT
binary	Binary	VARCHAR(max)



As a side effect, this table gives also a good overview about which data type in Power Query is compatible with which data type in SQL.

Second, create a stored procedure that creates the VIEW. The first step in the procedure is to LEFT JOIN the key-value pair table (KeyValue) and the lookup table for the data types (KeyType). If a data type can't be found, I default to NVARCHAR(3000). With function STRING_AGG, I concatenate the list of keys to a comma-separated list. I do this twice: once for a plain list, which is later used for the IN clause of the PIVOT keyword, and for a second one, I add TRY_CONVERT for the projection. The variables containing the result of the aggregation are then injected inside the CREATE OR ALTER VIEW definition:

```
CREATE or ALTER PROC [KeyValue].[CreateViewKeyValue] (
    @_Source varchar(50),
    @debug bit = 0
)
AS
BEGIN

SET NOCOUNT ON;

DECLARE
    @CRLF nvarchar(MAX) = CHAR(13)+CHAR(10),
    @cmd nvarchar(MAX),
    @ColumnNameKey nvarchar(max),
    @ColumnNamePivot nvarchar(max)

SELECT
    @ColumnNameKey = STRING_AGG([ColumnNameKey], ', ')
```

```

        WITHIN GROUP (ORDER BY [ColumnNameKey]),
@ColumnNamePivot = STRING_AGG([ColumnNamePivot], ', ')
        WITHIN GROUP (ORDER BY [ColumnNameKey])
FROM
    (SELECT
        DISTINCT
        CONVERT(varchar(max), [Key]) + @CRLF
        as [ColumnNameKey],
        CONVERT(varchar(max),
            N'TRY_CONVERT(' + [DataType] + N', ' + [Key]
            + N') AS ' + [Key] + @CRLF)
        as [ColumnNamePivot]
    FROM
        (
        SELECT DISTINCT
            kv.[_source]
            ,QUOTENAME(TRIM(kv.[key])) [Key]
            ,ISNULL(kt.[DataType], 'NVARCHAR(3000)') [DataType]
        FROM [dwh].[KeyValue] kv
        LEFT JOIN [KeyValue].[KeyType] kt ON kt.KeyType = kv.[Type]
        ) k
    ) x;

-- VIEW
SET @cmd=N'
CREATE OR ALTER VIEW [PowerBI].[KeyValue]
AS
(
    /** DO NOT MAKE ANY CHANGES DIRECTLY **/
    /** This code was generated **/
    SELECT
        p.ID [ID],
        + @ColumnNamePivot + N'
    FROM (SELECT [ID], [Key], [Value] FROM [dwh].[KeyValue]) kv
    PIVOT
        (MIN([Value]) FOR [Key] IN (
        + @ColumnNameKey + N'
    )
        ) as p
    )
'
if @debug = 1 exec [KeyValue].[Print] @cmd;
exec sp_executesql @stmt = @cmd

END

```

Now you've mastered a rather complex topic in SQL! This chapter concludes with how to combine the world of self-service and enterprises when it comes to a data warehouse.

Combining Self-Service and Enterprise BI

I assume that the idea of many programming languages developed in the 1960s and 1970s was that they should enable anyone who can write a natural language to control a computer. SQL is no exception here. The language is derived from the English language including the grammar. That's why it starts with the `SELECT` keyword: `SELECT "book" FROM "library."` (IntelliSense would appreciate if we would start with the `FROM` keyword instead, as, e.g., it is implemented in .NET's LINQ, because it would make it easier for IntelliSense to support you with helpful suggestions). A number of decades later, we can conclude that this plan did not work out. Only (some?) people from IT and a few power users are capable of writing SQL statements that produce useful answers from a relational database.

With that said, implementations in SQL are clearly part of an enterprise BI environment. This chapter has demonstrated that you can use the power of SQL to model and transform a data model so it makes the end user's life easy when working with tools like Power BI.

Key Takeaways

This chapter's use cases demonstrated that SQL is a very powerful language. Combined with the possibility to generate dynamic SQL, you can write resilient code and (semi-)automate a lot of processes. Therefore, I prefer a relational data warehouse (layer) for these tasks, if available, for my customers. Specifically, you learned how to apply SQL to solve the following challenges:

- You can use a combination of `DISTINCT` and `CASE` to build a static lookup table for the desired bins. You can directly join this table with the fact table. Alternatively, you can `UNION` a couple of `SELECT` statements to build a table containing the ranges for each bin. In SQL, you can write non-equi-joins to combine this information with the fact table. In Power BI, you can re-use this table, but must write a measure in DAX to implement the non-equi-join (see [Chapter 11](#)).
- For the budget problem, I prefer to create a bridge table, which simply contains the distinct values of the `Product Group` column. This table then has two one-to-many relationships, replacing to original many-to-many relationship.
- For the `TextColumn`, you learned that you can use `PIVOT`. This keyword needs a static list of the pivoted column's list, though. Therefore, I introduced you to dynamic SQL, where you can dynamically inject the list of columns into the `SELECT` statement.
- In the key-value pair scenario, applying `PIVOT` is necessary, too. Additionally, you need to take care of assigning the appropriate data types to the columns. In case a

data type per key is available, you learned how you can extend the dynamic SQL solution and inject the right data type conversions as well.

- SQL and relational databases are clearly most useful in an enterprise BI environment. The goal should not be to teach domain users the secrets of SQL but to build (with SQL) a data model that can easily be consumed by the domain experts in tools like Power BI Desktop, Power BI Report Builder, or Excel.

Last, but not least, I will show you how you can improve the performance of your data model with features in SQL.

Performance Tuning the Data Model with SQL

In Chapters 4 and 8, you learned several options for optimizing query response time through decisions about the storage mode of the tables in your Power BI data model. Physics mandates that you can obtain faster query response times by using more disk space—and the other way around. This is not only true for Power BI and Analysis Services but for all database systems, including relational databases.

In this chapter, I describe the options you have in relational databases to exchange storage space for query response time, and what you can do in the relational world to support query speed inside Power BI and Analysis Services.

Storage Modes

I title this section after Power BI's term *storage modes* to parallel other parts of the book even though it's typically not used when talking about relational databases. In relational databases, people usually talk about either *persisting* data in the database, or not. Persisting means to actually use disk space to store information in a certain form and shape, as opposed to only storing a query string. If the data is already stored in the right shape on disk, it only has to be transferred from the disk into memory and then sent over the network to the client. If only a query string is stored, everything I just mentioned has to be done as well, but on top of that, you need resources (CPU, memory, disk IO) to fetch the data from different tables, join it, and apply the one or another transformation to it.¹

¹ The file used for this section's examples is *401 Data vs Query.sql*.

In a relational database, you have the following options when it comes to storage: tables, indexes, compression, views, functions, and stored procedures.

Table

You can create a table in a relational database and then `INSERT`, `UPDATE`, and `DELETE` rows in it. By storing information in a table, you can apply complex transformations as soon as you insert a row into the table. Such a transformation would make updating the content of the table slower but improve the time it takes to query the information later.

Only when the table's content is to be read multiple times is it worth persisting the data into a table. If the sole purpose of the information is to feed an analytical system (like Power BI or Analysis Services), it's usually not worth the effort of triggering a job to persist the data or spending disk space on storage. Persisting the data will increase the overall time required to transfer the data from the data source into the analytical system.

If certain users (e.g., data scientists) or tools (which can send requests in SQL but not the analytical system's query language) are supposed to read regularly from the relational system, it's worth persisting the data and implementing an index strategy on the tables. Another use case that warrants persisting the data is if you're planning to implement a delta load based on partitioning. Only when the partitions between Power BI and the relational database are aligned can you expect a faster refresh of the Power BI semantic model. The same is true if you decided against importing the content of a table into Power BI, but keep the table in DirectQuery mode.

Index

An index on a table is like the index at the end of this book: it's extra space reserved for an ordered list of items used to speed up the time it takes to find certain information. While a book usually only contains a single index, a table can have several indexes. Microsoft's SQL databases (Azure SQL DB, Azure SQL Managed Instance, SQL Server, etc.) offer the following types of indexes:

- You can have one `CLUSTERED INDEX` per table. The special characteristic of a clustered index is that it doesn't use extra storage space, but puts the rows of the table itself in the order of the index column(s), like the order of the chapters in this book. The book's table of contents points out on which pages you'll find the information for a certain chapter or a section. As the rows of a table can be put into only one order, there can only be one `CLUSTERED INDEX` per table (just as a book can only be physically ordered according to one criteria).
- If you need to further improve queries (based on filters and groupings of other columns), you can add `NONCLUSTERED INDEXes` to the table. These use extra

storage space for index columns and a row identifier (like the book's index). The row identifier is like the page number. When using `NONCLUSTERED INDEX`s, the database management system needs to find the entry in the index first and then look up the row in the table if the query asks for nonindexed columns. You can avoid the need for the lookup by including further columns in the index. The index isn't sorted by those included columns. Operations that manipulate the content of the table will maintain the index automatically, but extra effort will be required to keep the index ordered and the content of the included columns up to date. Again, this exchanges storage space (and slower write operations) for improved query time.

When it comes to a physical implementation of a `CLUSTERED` or a `NONCLUSTERED` index, the most common form in relational databases is a so-called B-Tree. You can imagine a B-Tree as an upside-down tree with the individual rows of the table on the bottom (called leaves). If I ask you to guess a number between 1 and 100, you'll probably start by asking if it's greater or lower than 50. If it's bigger, then you might ask if it's greater or lower than 75, and so on. This search algorithm is called a binary search. You can imagine the structure of a B-Tree index as having been created in such a way as to support exactly this algorithm. As a rule of thumb when it comes to relational databases, you should create at least one index on a table (on the primary key of a table and its foreign keys).



Under the hood, a columnstore index is the same as the storage engine behind Power BI and Analysis Services. Any white paper on this index will explain how Power BI's *VertiPaq* is implemented. In theory, a DirectQuery connection on a data model hosted in a table with a clustered columnstore index should be as fast as the same data available in Import mode. Practically, small but crucial differences slow queries on a columnstore index: a columnstore index doesn't need to fit into memory (as with a data model in Power BI and Analysis Services) but can be paged to the disk. You can update, insert, and delete individual rows of a columnstore index, while you only can refresh a full partition of a Power BI and Analysis Services data model. This makes the Power BI implementation of this storage engine superior to the columnstore index of a relational database. So when you need to use DirectQuery, ensure that you apply the right index strategy. But import everything into a Power BI semantic model, if possible, to achieve the best query performance for your data model.

Technically, the *columnstore index* isn't an index but a completely different storage engine inside Microsoft's relational databases. As opposed to "normal" tables and indexes, which store the information of all columns for a single row physically

together, a columnstore splits up rows into their columns and stores the content of the columns physically together (as you might have guessed from the name). These blocks of similar data can be compressed very efficiently. You can create both a CLUSTERED and a NONCLUSTERED columnstore index on a table. As a rule of thumb, you should add a clustered columnstore index to all fact tables and big dimension tables in an analytical database. The high compression rate of this type of index (which typically squeezes the data to a tenth of its size without losing any information) is very welcome with these big tables.

Compression

For all tables where a clustered columnstore index isn't feasible (e.g., if it contains less than one million rows) you should turn on *page compression*. This not only saves you disk space, but usually decreases query time as well because the disk transfer is the bottleneck, for most servers, and the CPU tends to be underused. Compression will increase overhead to the CPU (for compressing and uncompressing the data), but less data will be transferred from and to the disk; thus, the overall query performance will improve.

View

A VIEW is simply a SELECT statement stored under a certain name in the database. It can be queried in the same way a table can. If you don't look at the metadata of the database object, you won't recognize a difference in syntax. However, query time might be slower if the SELECT statement of the definition of the VIEW contains complex queries or complex transformations, and/or the underlying tables are not indexed well enough. Especially when the sole purpose of the data warehouse (layer) is to provide a data model for Power BI or Analysis Services, there is no need to persist the data; it'll only be read once per refresh. Not even an index is necessary if you do only full refreshes. In such a case, putting all transformations into a VIEW is more than sufficient. Only when the data will be read more often (from different BI tools or by data analysts, data scientists, or similar) might it make sense to persist the data, as I pointed out when I explained the advantages and disadvantages of a table.



Serverless SQL pool in Azure Synapse Analytics offers a way to create a table as a SQL query. The result of the query will be persisted in a parquet file in the data lake. Storage costs for the data lake are very cheap compared to the storage costs in a relational database. Azure Synapse Serverless lets you use these inexpensive services in exchange for increased computing costs on the database side to collect (and transform) all the data from text files. The costs for the data at rest may be close to be negligible, but every query on such a table (e.g., to refresh your Power BI/Azure Analysis Services data model) will cost you.

Function

Different types of functions are available in T-SQL. Table-valued functions can be queried similarly to tables and views, with the difference that you need to write mandatory parentheses after the name. Inside the parentheses, you specify parameters, if applicable. I therefore tend to call table-valued functions “parametrized views.” The provided parameter(s) can help to optimize the query statement inside the definition of the function. Functions aren’t listed when you connect to the database from Power BI, but you need to provide a SQL statement for the data source, which then contains the function’s name in the FROM or JOIN portions.

Stored Procedure

A stored procedure can contain even very complex T-SQL code. Again, a procedure might have parameters, which might allow you to optimize the code inside. A procedure can’t be part of a query, but you need to execute it. Stored procedures are not listed when you connect to the database from Power BI, but you need to provide a T-SQL statement for the data source and then provide the name of the procedure plus its parameters (for example, `EXECUTE PowerBI.GetDates`).

In the next section, I introduce a different technique, which can speed up queries and your ETL process.

Partitioning

If the refresh of the Power BI/Analysis Services data model takes too much time, you should invest time in a good partitioning concept (either on the table you’re directly importing or on the underlying table of a view, function, or stored procedure). It’s important that the partition strategies of the relational table and the table in Power BI/Analysis Services are aligned (meaning that both use the same partition key). If you only partition the latter (or use a different partition key), then the refresh time might not improve at all because the full source table must still be scanned to provide the rows needed for the refreshed partition. It might even take longer—the refresh of every partition in Power BI may trigger a full scan of the whole table of Power BI’s data source.

Tables in DirectQuery mode will also benefit from partitioning when the partition key is also part of the queries filter, as whole partitions can be ruled out and less data needs to be scanned. If the partition key isn’t part of the filter, then you shouldn’t expect any drawbacks. A full scan of a partitioned table or a nonpartitioned table will take the same amount of time. So, implementing the right index is crucial to avoid full table scans also in the case of partitioned tables.

Partitioning will also help if you persist data in a relational data warehouse. I recommend using the timestamp of the creation of the row in the data source as the

partition key because it won't change. If you need to reload data for a certain period of time, you can easily switch out the partition in question into a table of its own, truncate this table, insert the new version of these rows into this table, and switch the partition back into the original table. All operations, except for the insert, are so-called *metadata operations*, which can be done very quickly (the duration is independent of the number of rows involved). Microsoft's [online documentation](#) will provide you with the necessary details to implement partitioning.

From a technical perspective, it's important to understand that you need to create a `PARTITION FUNCTION` that contains a list of values for the partition key and is used to assign a row of the table to a certain partition. Each value provided during the definition of the `PARTITION FUNCTION` is then either the left or the right border of the partition: `RANGE LEFT` means that the first partition will contain values lower than or equal to the first value of the `PARTITION FUNCTION` (the value is part of the partition to the left of it); `RANGE RIGHT` means that the first partition will contain values lower than the first value of the partition function (and the value mentioned in the `PARTITION FUNCTION` is part of the partition to the right of it). So, the type specifies whether the border's values are part of the partition to the left or right of the border. You always have one partition more than values listed in the `PARTITION FUNCTION`.

Then, you need to create a `PARTITION SCHEMA` that uses the `PARTITION FUNCTION`. To partition a table, you need to create it `ON` this `PARTITION SCHEMA`. Let's see these commands in action.²

First, create a `PARTITION FUNCTION` with the name `pfOrderDate` that accepts partition keys of data type `datetime2(0)`, which means dates including timestamps with a precision to the second. This data type must match the data type of the column you use as a partition key. The function has three values. All partition keys before September 1, 2023 are assigned to the first partition (due to `RANGE RIGHT` being implemented, September 1, 2023 is part of the partition to the right of it):

```
CREATE PARTITION FUNCTION pfOrderDate (datetime2(0))
AS RANGE RIGHT FOR VALUES
('2023-09-01', '2023-10-01', '2023-11-01');
```

As the function defines three borders, there will be four partitions. The `RANGE RIGHT` or `RANGE LEFT` option only defines whether the value of the border itself is part of the partition to the right or left of the border ([Figure 20-1](#)).

Next, create a `PARTITION SCHEME` named `psOrderDate`, which references the newly created `PARTITION FUNCTION` with the name `pfOrderDate`. In the example, all partitions will be hosted in the `PRIMARY` file group of the database.

² The file used for this section's examples is [403 Partitioning.sql](#).

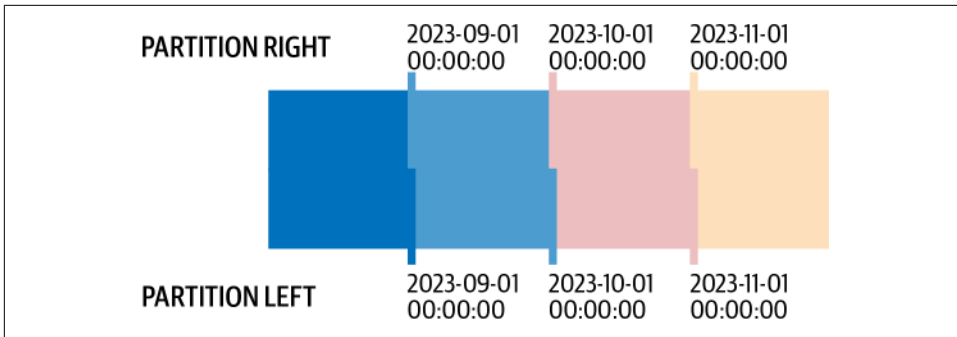


Figure 20-1. Partition *RANGE RIGHT* versus *RANGE LEFT*



Technically, it makes no difference whether you choose *RANGE RIGHT* or *RANGE LEFT*, but if your partition key is somehow related to a point in time (date or `datetime2`), I'd strongly recommend opting for *RANGE RIGHT*. You want to have all rows for a day or month in the same partition. With *RANGE LEFT*, rows for midnight/the first day of the month would be in a different partition than rows for the same day but at a later point in time, or later in the month, as you can see in [Figure 20-1](#).

Alternatively, you could provide a list of the `FILEGROUPS`. The first partition would be then hosted in the first `FILEGROUP`, the second partition in the second `FILEGROUP`, etc. In the definition of a `FILEGROUP`, you specify in which physical file on which physical drive the data is stored. This enables you to put, e.g., the partitions containing the newest data, which are updated and queried often, on a fast SSD drive, while the partitions for older years, which are never updated and queried very rarely, may reside on cheaper storage. You can find more information about managing file groups in the [Microsoft documentation](#):

```
CREATE PARTITION SCHEME psOrderDate
AS PARTITION pfOrderDate
ALL TO ('PRIMARY') ;
```

Last but not least, create a table `dbo.PartitionTable`, which you put ON the newly created `PARTITION SCHEMA` with the name `psOrderDate`. Pass `OrderDate` as the parameter for the `PARTITION SCHEMA`. For each row, the content of `OrderDate` will decide in which partition the row will be stored—it is your partition key. Having this column as the first (or only) column of the `PRIMARY KEY` and any index will further speed up queries, as they are then aligned with the partition:

```
CREATE TABLE partitioning.FactResellerSales (
    [OrderDate] datetime2(0),
    [SalesAmount] decimal(19, 2)
)
ON psOrderDate (OrderDate) ;
```

Let's insert some rows and then find out in which partitions they landed:

```
INSERT INTO partitioning.FactResellerSales
SELECT OrderDate, SalesAmount
FROM PowerBI.FactResellerSales
```

Group the rows of this table by OrderDate and count the rows:

```
SELECT
    OrderDate,
    COUNT(*) [RowCount]
FROM
    PowerBI.FactResellerSales
GROUP BY
    OrderDate
ORDER BY
    OrderDate;
```

OrderDate	RowCount
...	...
2023-07-01 00:00:00.000	2,076
2023-08-01 00:00:00.000	2,177
2023-09-01 00:00:00.000	1,797
2023-10-01 00:00:00.000	2,847
2023-11-01 00:00:00.000	3,004

The expectation is that a lot of rows will be added to the first partition (which contains everything before September 1, 2023). But only 1,797 rows will be put into the second partition (on or after September 1, 2023); 2,847 rows will go into the third partition (on or after October 1, 2023); and 3,004 rows will go into the fourth partition (on or after November 1, 2023). **Example 20-1** proves this successfully. It joins metadata tables from the database's sys schema (schemas, tables, indexes, partitions, filegroups (twice), and destination_data_space) and then shows the schema's name, the table's name, the partition number, the file group, and the row count.

Example 20-1. Count rows per partition

```
-- Count rows per partition
SELECT
    s.[name] AS SchemaName
    , t.[name] AS TableName
```

```

, ds.[name] AS PartitionScheme
, p.partition_number AS PartitionNumber
, COALESCE(f.[name], d.[name]) AS [FileGroup]
, p.[rows] AS [RowCount]
FROM
sys.schemas AS s
INNER JOIN sys.tables AS t ON
    t.schema_id = s.schema_id
INNER JOIN sys.partitions AS p ON
    p.object_id = t.object_id
INNER JOIN sys.indexes AS i ON
    i.[object_id] = p.[object_id] AND
    i.index_id = p.index_id
LEFT JOIN sys.index_columns AS ic ON
    ic.[object_id] = i.[object_id] AND
    ic.index_id = i.index_id
LEFT JOIN sys.columns AS c ON
    c.[object_id] = ic.[object_id] AND
    c.column_id = ic.column_id
LEFT JOIN sys.data_spaces AS ds ON
    ds.data_space_id = i.data_space_id
LEFT JOIN sys.data_spaces AS ds ON
    ds.data_space_id = i.data_space_id
LEFT JOIN sys.partition_schemes AS ps ON
    ps.data_space_id = ds.data_space_id
LEFT JOIN sys.partition_functions AS pf ON
    pf.function_id = ps.function_id
LEFT JOIN sys.filegroups AS f ON
    f.data_space_id = i.data_space_id
LEFT JOIN sys.destination_data_spaces AS dds ON
    dds.partition_scheme_id = i.data_space_id AND
    dds.destination_id = p.partition_number
LEFT JOIN sys.filegroups AS d ON
    d.data_space_id = dds.data_space_id
LEFT JOIN sys.partition_range_values AS prv_left ON
    ps.function_id = prv_left.function_id AND
    prv_left.boundary_id = p.partition_number - 1
LEFT JOIN sys.partition_range_values AS prv_right ON
    ps.function_id = prv_right.function_id AND
    prv_right.boundary_id = p.partition_number
WHERE
s.[name] = 'partitioning' AND
t.[name] IN ( 'FactResellerSales', 'FactResellerSales_STAGE' ) AND
t.[type] = 'U' AND
i.index_id IN (0, 1)
ORDER BY
s.[name]
, t.[name]
, p.index_id
, p.partition_number;

```

SchemaName	TableName	PartitionScheme	PartitionNumber	FileGroup	RowCount
partitioning	FactResellerSales	psOrderDate	1	PRIMARY	53,207
partitioning	FactResellerSales	psOrderDate	2	PRIMARY	1,797
partitioning	FactResellerSales	psOrderDate	3	PRIMARY	2,847
partitioning	FactResellerSales	psOrderDate	4	PRIMARY	3,004

As I've explained, a partitioned table can be queried faster, if the partition key is part of the filter. Even if no index is available to satisfy the search condition, only the partition needs to be scanned for the rows, instead of the full table. Partitioning is also helpful when you need to reload data into the table for a specific day. Without partitions per day, you would need to either DELETE and INSERT all the rows for this day (a full load) or implement a logic to remove outdated rows, insert new rows, and update changed rows (a delta load). All these operations can take a while to complete when executed on a large table.

With partitioning, you can tremendously speed up the process of reloading the data for one day (assuming that you use a date as the partition key of your table). First, create an empty copy of the table, which will host the rows that should be updated. In the following code, the INTO clause creates the new table (with suffix _STAGE) and selects the TOP 0 rows (no data) for all columns (*). You need to make sure that the newly created table is put on the same file group as the partition scheme:

```
DROP TABLE IF EXISTS partitioning.FactResellerSales_STAGE;
SELECT TOP 0 *
INTO partitioning.FactResellerSales_STAGE ON [PRIMARY]
FROM partitioning.FactResellerSales;
```

Then, SWITCH a specific partition out of the original table (partitioning.FactResellerSales) and into the stage table. This operation is incredibly fast because no data is moved; only the metadata for the partition's rows is changed. The partition information for the rows is changed only to indicate that they are now part of the stage table. The following example specifies to SWITCH partition 3, which is made up of the 2,847 rows from October 1:

```
ALTER TABLE partitioning.FactResellerSales
SWITCH PARTITION 3
TO partitioning.FactResellerSales_STAGE;
```

If you rerun the query to count the rows per partition (Table 20-0), you will see that partition 3 of FactResellerSales has a row count of 0, but the only partition available in FactResellerSales_STAGE has a row count of 2,847:

SchemaName	TableName	PartitionScheme	PartitionNumber	FileGroup	RowCount
partitioning	FactResellerSales	psOrderDate	1	PRIMARY	53,207
partitioning	FactResellerSales	psOrderDate	2	PRIMARY	1,797
partitioning	FactResellerSales	psOrderDate	3	PRIMARY	0
partitioning	FactResellerSales	psOrderDate	4	PRIMARY	3,004
partitioning	FactResellerSales_STAGE	PRIMARY	1	PRIMARY	2,847

Now it's easy to work with the data for October 1 because all the rows are sitting in a much smaller table. For a full load, you can execute a `TRUNCATE TABLE partitioning.FactResellerSales` to erase the content (as a, once again, fast metadata operation) and insert the source's data into it again.

If you're finished bringing the rows for October 1 up-to-date, you can `SWITCH` the full table back into `FactResellerSales`:

```
ALTER TABLE partitioning.FactResellerSales_STAGE SWITCH TO partitioning.
FactResellerSales PARTITION 3;
```

Unfortunately, you'll receive an error message asking for a mandatory `CHECK CONSTRAINT`, which must be present on the stage table:

```
ALTER TABLE SWITCH statement failed. Check constraints of source table
'AdventureWorksDW.partitioning.FactResellerSales_STAGE' allow values that
are not allowed by range defined by partition 3 on target table
'AdventureWorksDW.partitioning.FactResellerSales'.
```

The mandatory check constraint must reflect the rule that allows a row to be part of partition 3, which is that the `OrderDate` must be after or equal to October 1 and before November 1:

```
ALTER TABLE partitioning.FactResellerSales_STAGE
WITH CHECK
ADD CONSTRAINT CK_FactResellerSales_STAGE_OrderDate
CHECK (
    OrderDate IS NOT NULL AND
    OrderDate >= {d'2023-10-01'} AND
    OrderDate < {d'2023-11-01'}
)
```

With the following addition to the projection of the query in [Example 20-1](#), you get the text for the `CHECK` constraint listed in the query result, derived from the partition's definition:

```
, QUOTENAME(c.name) + ' IS NOT NULL AND ' +
CASE pf.boundary_value_on_right WHEN 1 THEN
-- 'RIGHT'
ISNULL(QUOTENAME(c.name) + ' >= ' +
convert(varchar, prv_left.[value], 126) + ''', ''') +
case when p.partition_number NOT IN (1, MAX(p.partition_number) OVER())
```

```

then ' AND '
else ''
end +
ISNULL(QUOTENAME(c.name) + ' < ' +
convert(varchar, prv_right.[value], 126) + ''', '')
ELSE
-- 'LEFT'
ISNULL('[PartitionKey] > ' +
convert(varchar, prv_left.[value], 126) + ''', '') +
case when p.partition_number NOT IN (1, MAX(p.partition_number) OVER())
then ' AND '
else ''
end + ISNULL(QUOTENAME(c.name) + ' <= ' +
convert(varchar, prv_right.[value], 126) + ''', '')
END as CheckConstraint

```

Here's the result for this additional column, which you can just copy and paste into the ALTER TABLE statement:

...	TableName	PartitionNumber	...	CheckConstraint
...	FactResellerSales	1	...	[OrderDate] IS NOT NULL AND [OrderDate] < '2023-09-01'
...	FactResellerSales	2	...	[OrderDate] IS NOT NULL AND [OrderDate] >= '2023-09-01' AND [OrderDate] < '2023-10-01'
...	FactResellerSales	3	...	[OrderDate] IS NOT NULL AND [OrderDate] >= '2023-10-01' AND [OrderDate] < '2023-11-01'
...	FactResellerSales	4	...	[OrderDate] IS NOT NULL AND [OrderDate] >= '2023-11-01'

If you now retry switching the partition back into FactResellerSales, it will succeed. If you didn't change the content of the stage table, the content of FactResellerSales should be back to the original state:

SchemaName	TableName	PartitionNumber	FileGroup	RowCount
partitioning	FactResellerSales	1	PRIMARY	53,207
partitioning	FactResellerSales	2	PRIMARY	1,797
partitioning	FactResellerSales	3	PRIMARY	2,847
partitioning	FactResellerSales	4	PRIMARY	3,004
partitioning	FactResellerSales_STAGE	1	PRIMARY	0

Instead of speeding up the access to the data itself, you can calculate intermediate results on an aggregated level and take the data from there, as the next section explains.

Pre-Aggregating

Providing Power BI with pre-aggregated data can be done with any of the relational storage modes that “[Storage Modes](#)” on [page 417](#) discusses: (indexed) table, view, function, or stored procedure. With all of these, you can either Import or access in DirectQuery mode from Power BI and Analysis Services tabular.

To create an aggregated version of a table, all you need to do is to specify the granularity (basically, the dimension keys) and aggregation functions (typically COUNT or SUM) on the numeric values.³ Watch out for the GROUP BY clause, in which you need to put all the columns that aren’t wrapped inside an aggregation function:

```
CREATE OR ALTER VIEW agg.FactResellerSalesAgg AS
SELECT
    OrderDate,
    COUNT(*) SalesCount,
    SUM(SalesAmount) SalesAmount
FROM
    PowerBI.FactResellerSales
GROUP BY OrderDate
```

After you’ve added this view as an additional table to your data model, do not forget to tell Power BI that this table is an aggregated version of the already existing table FactResellerSales (“[Pre-Aggregating](#)” on [page 166](#)), or make sure to include logic in your measures to use the aggregated table, where appropriate (“[Pre-Aggregating](#)” on [page 235](#)).



“[Calculations](#)” on [page 376](#) explains that not all operations’ results are aggregable. Aggregations like a distinct count or a percentage are non-additive, and should therefore not be added to an aggregation table.

³ This section uses the [402 Aggregated Facts.sql](#) file for examples.

Key Takeaways

This chapter talked about both how to increase the speed of operations in a relational database and how to support speedy operations in Power BI through a well-thought-out design of your database objects. In summary, you learned the following:

- In a relational database, information can be provided either persisted in a table or as a query stored inside a view, function, or stored procedure.
- Information stored in a table should definitely be indexed. You learned about the principles of how an index works. And you learned that a columnstore index is not truly an index but supports a query engine inside a relational database, which is basically identical to the query engine of Power BI and Analysis Services tabular.
- Partitioning is crucial for both big tables in a relational database and big tables in Power BI and Analysis Services. If you decide to use partitioning, make sure to use the identical column to partition in the relational world and in Power BI/Analysis Services, to align the partitions.
- You can pre-aggregate information in the relational layer with the help of the GROUP BY clause and aggregation functions.

Epilogue

Congratulations—you made it through to the last page! Thank you so much for spending your precious time reading this book. I hope you enjoyed reading it as much as I enjoyed writing it. Before I let you go, here’s a quick summary of the most important things when it comes to data modeling for Power BI:

- You should sit down and understand your data and the business’s needs before connecting to the first data source. Data profiling in Power Query can help you understand the data; it shows descriptive statistics for each column.
- Bringing the source’s information into a dimensional model is crucial when it comes to Power BI. Any shortcut taken here will come back to haunt you at a later stage. (I speak from experience!) Many-to-many relationships and bi-directional filters exist only for very special use cases. I build many data models solely based on one-to-many relationships and single-directed filters.
- Make sure that every table is connected to at least one other table. Unrelated tables are useful only in special cases (like when you need a non-equi-join). If you can’t connect a table to other tables and the table is not used in connection with other tables of the semantic model, consider removing this table from this semantic model and building a different semantic model for it.
- Push all necessary transformations as early as possible up the data chain. Use Power Query over DAX for this task. If you can, convince the people providing you with Excel sheets to bring them into the necessary shape, or ask for access to the database the Excel sheets are based on. Even better, talk to the right people in your organization to set up a data warehouse layer, where all the transformation “magic” should happen. This way, you make all the work more re-usable throughout your organization.

- When it comes to all sorts of calculations (especially semi-additive and non-additive ones), you need to develop DAX measures. Don't create calculated columns. Don't add the results of such calculations as columns during the data transformation process.
- When you discover obstacles, like slow performance or very complicated DAX calculations, take a step back and reevaluate if your data model really is in a star schema.
- You will get the best possible report speed by importing data to Power BI. In case you discover problems with data size or refresh time, try out aggregation, dual, hybrid tables, or a composite model before changing a whole data model to DirectQuery mode. DirectQuery is only the second-best option.
- Mastering data modeling takes time and effort. Practice is king. Failing is part of the journey!

And last but not least, the most important thing to remember: the goal of the data model is to make the report creator's life easy!

A

- accumulated snapshot fact table, 36
- ADDCOLUMNS (DAX function), 204
- additive calculations, 46
- AdventureWorksDW, installing, xxxiii-xxxv
- aggregable columns, 229
- aggregated fact table, 35
- aggregation
 - in DAX calculations, 195
 - pre-aggregating
 - in DAX, 235
 - in Power BI data model, 166-167
 - in Power Query, 329-331
 - in SQL, 429
- aggregation-aware measures (DAX), 236
- ALL (DAX function), 96
- Analysis Server tabular, xxiv
- anchor query, 397
- anti-joins
 - basics, 17
 - SQL implementation, 347-348
- anti-patterns, 40-41
- Azure Cognitive Services, 313
- Azure Data Studio, installing, xxxiii
- Azure SQL DB, xxviii, xxxiii

B

- binning
 - basics, 58-60
 - adding a column to a fact table, 58
 - creating a lookup table, 58
 - defined, 58
 - describing the ranges of bins, 59
 - with DAX
 - lookup tables, 216-217
 - range tables, 217-220

- with Power BI, 134
 - lookup tables, 134
 - range tables, 135
- with Power Query and M, 298-308
 - creating bin range table in M, 307-308
 - creating bin table by hand, 298
 - creating bin table in M, 301-307
 - deriving bin table from facts, 299
- with SQL, 399-403
 - deriving lookup table from facts, 400-401
 - generating lookup table, 401
 - range tables, 402-403
- BLANK value, 202
 - (see also NULL value)
- bridge tables
 - defined, 28
 - using in DAX, 138
- budget use cases
 - DAX, 220-222
 - in SQL, 403
 - Power BI, 135-139
 - Power Query and M, 308-313
- building a data model, 43-56
 - calculations, 46
 - denormalizing, 45-46
 - flags and indicators, 47
 - normalizing, 44
 - role-playing dimensions, 48
 - slowly changing dimensions, 49-53
 - type 0: retaining original dimensions, 49
 - type 1: overwriting, 50

- type 2: adding a new row, 51-52
 - type 3: adding new attributes, 52
 - type 4: adding mini-dimensions, 53
 - types 5, 6, 7: combining changes to dimensions, 53
 - time and date, 47-48
- ## C
- CALCULATE (DAX function), 198-200, 206, 220, 224
 - calculations
 - in data model, 46
 - in DAX, 190-200
 - re-creating a column as a DAX measure, 196-197
 - semi-additive, 195
 - simple aggregations, 195
 - time-intelligence, 198-200
 - in Power BI, 112-116
 - in Power Query and M, 273
 - in SQL, 376-379
 - CALENDAR (DAX function), 203
 - CALENDARAUTO (DAX function), 203
 - cardinality
 - basics, 10
 - Power BI, 95-96
 - chasm traps, 21-23, 352-354
 - clustered index, 418
 - columns
 - adding to fact table, 58
 - Power Query and M
 - identifying, 261-270
 - normalizing distribution, 259
 - normalizing profiles, 260
 - normalizing quality, 258
 - columnstore index, 419
 - common table expression (CTE)
 - collecting information with, 54
 - dividing code with, 397
 - composite keys
 - Power BI and, 92, 94
 - primary keys and, 8
 - composite models, 151, 168
 - compression, 420
 - CONCATENATE (DAX function), 180
 - corporate information factory database, 39
 - COUNTROWS (DAX function), 219
 - CROSS JOIN (SQL operator), 348
 - cross joins, 19
 - CROSSJOIN (DAX function), 184, 205
- ## D
- data marts, 39
 - data modeling (basics), 3-42
 - as representation, not replication, 4-5
 - as underestimated task with Power BI Desktop, xviii
 - basic components, 5-11
 - cardinality, 10
 - entity, 5
 - foreign keys, 9
 - primary keys, 7
 - relationships, 6-7
 - surrogate keys, 8
 - tables, 6
 - combining tables, 11-27
 - entity relationship diagrams, 25-27
 - join path problems, 20-25
 - joins, 13-19
 - set operators, 11
 - data modeling options, 28-39
 - anti-patterns, 40-41
 - data vaults, 40-41
 - dimensional modeling, 33-35
 - extract, transform, load (ETL), 36-38
 - granularity, 35-36
 - normal forms, 29-33
 - Ralph Kimball's and Bill Inmon's contributions, 38
 - single table for storage, 28
 - types of tables, 28
 - definition/brief history, xx
 - modeling options, 28-39
 - anti-patterns, 40-41
 - data vaults, 40-41
 - data vaults, 40-41
 - data warehouses (DWHs), 33
 - DATATABLE (DAX function), 202, 217
 - DAX (basics), xxvi, 175-186
 - adding calculations with, xvii
 - combining queries, 180-185
 - extract, transform, load (ETL), 185
 - joins, 182-184
 - set operators, 180-182
 - creating relationships, 179
 - CROSSJOIN, 184
 - DAX/data model relationship, 175-185
 - NATURALINNERJOIN, 184

- NATURALLEFTOUTERJOIN, 183
 - primary keys, 180
 - tables, 176-179
 - DAX (data model building), 187-214
 - calculations, 190-200
 - re-creating as DAX measure, 196-197
 - semi-additive calculations, 195
 - simple aggregations for additive calculations, 195
 - time-intelligence calculations, 198-200
 - denormalizing, 190
 - flags and indicators, 200-203
 - IF function, 200
 - lookup tables, 202
 - SWITCH function, 201
 - treating BLANK values, 202
 - hierarchies, 210-213
 - normalizing, 187-189
 - role-playing dimensions, 206-207
 - slowly changing dimensions, 207-210
 - time and date, 203-205
 - DAX (performance tuning), 235-238
 - aggregation-aware measures, 236
 - pre-aggregating, 235
 - storage mode, 235
 - DAX (real-world examples), 215-233
 - binning, 216-220
 - lookup tables, 216-217
 - range tables, 217-220
 - budget use case, 220-222
 - combining self-service/enterprise BI, 232
 - key-value pair tables, 229-231
 - multi-language model, 222-228
 - denormalizing, 45-46
 - in DAX, 190
 - in Power BI, 109-112
 - in Power Query and M, 270-273
 - in SQL, 375-376
 - dimension table, 34, 139
 - dimensional modeling, 33-35, 107
 - dimensions, in data models
 - role-playing dimensions, 48
 - slowly changing dimensions, 49-53
 - type 0: retaining original dimensions, 49
 - type 1: overwriting, 50
 - type 2: adding a new row, 51-52
 - type 3: adding new attributes, 52
 - type 4: adding mini-dimensions, 53
 - types 5, 6, 7: combining changes to dimensions, 53
 - DirectQuery, 134, 151, 154-160
 - and composite models, 168
 - and dual mode, 169
 - and hybrid tables, 170
 - and pre-aggregation, 166-167
 - dual mode
 - in Power BI data model, 169
 - DWHs (data warehouses), 33
- ## E
- enterprise BI, combining with self-service BI
 - basics, 67
 - in DAX, 232
 - in Power BI, 150-152
 - in Power Query and M, 324
 - in SQL, 414
 - enterprise data bus, 39
 - entities
 - basics, 5
 - relationship diagrams, 25-27
 - entity relationship diagrams (ERDs), 25-27, 100
 - entity tables, 28
 - equi-joins, 13, 350
 - ERDs (entity relationship diagrams), 25-27, 100
 - EXCEPT (DAX operator), 182
 - EXCEPT (SQL operator), 344
 - extract, transform, load (ETL)
 - basics, 36-38
 - in DAX, 185
 - in Power BI, 108
 - in Power Query, 256
 - in SQL, 357-359
- ## F
- fact table
 - adding columns, 58
 - in data models, 34
 - fan traps, 24-25, 354-356
 - flags and indicators, 47
 - in DAX, 200-203
 - IF function, 200
 - lookup table, 202
 - SWITCH function, 201
 - treating BLANK values, 202
 - in Power Query and M, 275-279
 - in SQL, 379-383
 - foreign keys

- in data models, 9
- in Power BI, 94
- in SQL, 340-341

FUNCTION (SQL), 370-371

functional dependent columns, 31

G

GENERATESERIES (DAX function), 205

granularity, defined, 35-36

H

HASONEVALUE (DAX function), 229

hierarchies

- in DAX, 210-213
- in Power BI, 129-130
- in Power Query and M, 286-294
- in SQL, 395-397

hybrid tables, 170

I

IF (DAX function), 200

incremental refresh, 329

indexed view, 362

indexing/indexes

- clustered/nonclustered, 336, 418
- in SQL storage modes, 418-419
- metadata and, 70

Inmon, Bill, 38

inner joins, 14

INSCOPE (DAX function), 212

INTERSECT (DAX operator), 182

INTERSECT (SQL operator), 343

islands, 158

J

join path problems, 20-25

- basics, 20-25
- chasm traps, 21-23, 352-354
- fan traps, 24-25, 354-356
- loops, 20-21, 351
- Power BI tables, 98
- SQL, 351-356

join predicates, 13

joins, 13-19

- Power BI tables, 97
- Power Query, 252
- SQL, 344-350
- anti-join implementation, 347-348

CROSS JOIN, 348

- equi-joins, 350

INNER JOIN, 344

- natural join implementation, 350
- non-equi-joins, 350

OUTER JOIN, 345-346

- self joins, 349

K

key-value pair tables, 65-66

- DAX, 229-231
- in SQL, 408-413
- Power BI, 149-150
- Power Query and M, 315-324
 - using M code, 319-320
 - using the GUI, 316-319
 - writing an M function, 320-324

Kimball, Ralph, 38, 49

L

LEFT JOIN, 367

live connection, 151, 155

lookup tables

- creating, 58
- defined, 28
- in DAX, 202
- in DAX binning, 216-217
- in Power BI binning, 134

loops, 20-21, 351

M

M (Power Query mashup language), 241, 244

- (see also Power Query and M entries)

materialized view, 362

metadata, in Power BI multi-language model, 143-145

mini-dimensions, 53

Model view, 75-108

- basics, 75-78
- entity relationship diagrams in, 100
- tables in, 78-88

multi-fact models, 60-63

- (see also budget use cases)

multi-language models

- basics, 63-64
- DAX, 222-228
- Power BI, 139-148

- dimension table for available languages, 139
- metadata translations, 143-145
- numerical content, 142
- text-based content, 141
- UI of Power BI Desktop (standalone), 146
- UI of Power BI Desktop (Windows store), 147
- UI of Power BI report server, 148
- visual elements, 140
- Power Query and M, 313-315
- SQL, 404-408

N

- natural hierarchy, 45, 53
- natural joins, 13, 350
- NATURALINNERJOIN (DAX function), 184
- NATURALLEFTOUTERJOIN (DAX function), 183
- non-additive calculations
 - as DAX measure, 196
 - in data models, 46
- non-aggregable columns, 229
- non-equi-joins, 350
- nonclustered index, 418
- normal forms, 29-33, 106
- normalizing, 44
 - in DAX, 187-189
 - in Power BI, 109-112
 - in Power Query and M, 258-270
 - column distribution, 259
 - column profile, 260
 - column quality, 258
 - identifying columns to normalize, 261-270
 - in SQL, 362-374
 - creating a function, 370-371
 - creating a PROCEDURE, 371-374
 - creating a VIEW, 369
 - persisting into a table, 367-368
- NULL value, 389-392
 - (see also BLANK value)
- numerical content, in Power BI multi-language model, 142

O

- one big table (OBT), 44, 111

- Online Analytical Processing (OLAP) database, 43
- Online Transactional Processing (OLTP) database, 43
- organigrams, 129
- OUTER JOIN (SQL operator), 345-346
- outer joins, 15, 93, 97

P

- paginated reports, 41
- parent-child hierarchy, 53
- partition key, choosing, 161
- partitioning
 - in Power BI data model, 160-164
 - in Power Query, 328-329
 - in SQL, 421-428
- PATHLENGTH (DAX function), 212
- performance tuning
 - basics, 69-71
 - DAX, 235-238
 - aggregation-aware measures, 236
 - pre-aggregating, 235
 - storage mode, 235
 - in SQL, 417-430
 - partitioning, 421-428
 - pre-aggregating, 429
 - storage modes, 417-421
- Power BI, 153-171
 - composite models, 168
 - dual mode, 169
 - hybrid tables, 170
 - partitioning, 160-164
 - pre-aggregating, 166-167
 - storage mode, 153-160
- Power Query, 327-332
 - partitioning, 328-329
 - pre-aggregating, 329-331
 - storage mode, 327
- periodic snapshot fact table, 36
- persisting data, 417
- PIVOT (SQL keyword)
 - for key-value pairs, 409-413
 - for multi-language models, 405-408
- Power BI (basics), 75-108
 - advantages of data model, xxv
 - cardinality, 95-96
 - combining tables, 97-100
 - entity relationship diagrams, 100
 - join path problems, 98

- joins, 97
- set operators, 97
- data modeling options, 101-108
 - dimensional modeling, 107
 - extract, transform, load (ETL), 108
 - granularity, 107
 - normal forms, 106
 - single table for storage, 103-106
 - types of tables, 101-103
- overview of data model, 75-78
- overview of tools, xx-xxv
- Power BI (data model building), 109-131
 - calculations, 112-116
 - hierarchies, 129-130
 - normalizing/denormalizing, 109-112
 - role-playing dimensions, 123-126
 - slowly changing dimensions, 127-128
 - time and date, 116-122
 - marking the date table, 121
 - turning off auto date/time, 116-121
- Power BI (performance tuning), 153-171
 - composite models, 168
 - dual mode, 169
 - hybrid tables, 170
 - partitioning, 160-164
 - pre-aggregating, 166-167
 - storage mode, 153-160
- Power BI (real-world examples)
 - binning, 134
 - lookup tables, 134
 - range tables, 135
 - budget use case, 135-139
 - combining self-service/enterprise BI, 150-152
 - key-value pair tables, 149-150
 - multi-language model, 139-148
 - dimension table for available languages, 139
 - metadata translations, 143-145
 - numerical content, 142
 - text-based content, 141
 - UI of Power BI Desktop (standalone), 146
 - UI of Power BI Desktop (Windows store), 147
 - UI of Power BI Report Server, 148
 - UI of the Power BI service, 148
 - visual elements, 140
- Power BI Desktop
 - installing, xxxiii
 - overview, xx
- Power BI Report Builder, xxxiii
- Power BI Report Server, xxii, 148
- Power BI service
 - incremental refresh, 329
 - overview, xxi
 - parameters, 255
 - UI, 148
- Power Query (basics), xvii, xxvii, 241-256
 - basic components, 244-251
 - primary keys, 248
 - relationships, 248
 - surrogate keys, 249-251
 - tables versus queries, 244-247
 - combining queries, 251-256
 - extract, transform, load (ETL), 256
 - joins, 252
 - query dependencies, 253
 - set operators, 251
 - types of queries, 255-256
 - data model, 242-244
- Power Query (performance tuning), 327-332
 - partitioning, 328-329
 - pre-aggregating, 329-331
 - storage mode, 327
- Power Query and M (data model building), 257-295
 - calculations, 273
 - denormalizing, 270-273
 - flags and indicators, 275-279
 - hierarchies, 286-294
 - normalizing, 258-270
 - column distribution, 259
 - column quality, 258
 - identifying columns to normalize, 261-270
 - role-playing dimensions, 284-285
 - slowly changing dimensions, 285
 - time and date, 279-284
- Power Query and M (real-world examples), 297-325
 - binning, 298-308
 - creating bin range table in M, 307-308
 - creating bin table by hand, 298
 - creating bin table in M, 301-307
 - deriving bin table from facts, 299
 - budget use case, 308-313
 - combining self-service/enterprise BI, 324

- key-value pair tables, 315-324
 - using M code, 319-320
 - using the GUI, 316-319
 - writing an M function, 320-324
- multi-language model, 313-315
- pre-aggregating
 - in DAX, 235
 - in Power BI data model, 166-167
 - in Power Query, 329-331
 - in SQL, 429
- primary keys
 - basics, 7
 - in DAX, 180
 - in Power BI, 94
 - in Power Query, 248
 - in SQL, 338
- PROCEDURE (SQL), 371-374

Q

- queries
 - combining (DAX), 180-185
 - extract, transform, load (ETL), 185
 - joins, 182-184
 - set operators, 180-182
 - combining (Power Query)
 - extract, transform, load (ETL), 256
 - joins, 252
 - query dependencies, 253
 - set operators, 251
 - types of queries, 255-256
 - combining (SQL), 341-359
 - entity relationship diagrams, 356
 - extract, transform, load (ETL), 357-359
 - join path problems, 351-356
 - joins, 344-350
 - set operators, 341-344
 - in Power Query, 244-247
- query folding, 328

R

- range tables
 - DAX, 217-220
 - in SQL, 402-403
 - Power BI, 135
 - Power Query and M, 307-308
- real-world examples
 - basics, 57-68
 - binning
 - basics, 58-60

- DAX, 216-220
 - in SQL, 399-403
- Power BI, 134
- Power Query and M, 298-308
- budget use case
 - basics, 60-63
 - DAX, 220-222
 - Power BI, 135-139
 - Power BI and SQL, 403
 - Power Query and M, 308-313
- combining self-service/enterprise BI
 - basics, 67
 - DAX, 232
 - in SQL, 414
 - Power BI, 150-152
 - Power Query and M, 324
- key-value pair tables
 - basics, 65-66
 - DAX, 229-231
 - in SQL, 408-413
 - Power BI, 149-150
 - Power Query and M, 315-324
- multi-language model
 - basics, 63-64
 - DAX, 222-228
 - in SQL, 404-408
 - Power BI, 139-148
 - Power Query and M, 313-315
- using DAX, 215-233
- using Power BI, 133-152
- using Power Query and M, 297-325
- using SQL, 399-415
- RELATED (DAX function), 190, 202
- relational data models, 335-360
 - basic components, 336-341
 - foreign keys, 340-341
 - primary keys, 338
 - relationships, 338
 - surrogate keys, 339
 - tables, 336-338
- combining queries, 341-359
 - entity relationship diagrams, 356
 - extract, transform, load (ETL), 357-359
 - join path problems, 351-356
 - joins, 344-350
 - set operators, 341-344
- relationships between entities
 - basics, 6-7
 - in DAX, 179

- in Power BI, 88-93
- in Power Query, 248
- in SQL, 338
- relationship diagrams, 25-27

REMOVEFILTERS (DAX function), 96, 104

role-playing dimensions

- basics, 48
- in DAX, 206-207
- in Power BI, 123-126
- in Power Query and M, 284-285
- in SQL, 385-387

rows, adding new, 51-52

S

SELECT statements (SQL), xxviii

SELECTEDMEASURE (DAX function), 199

SELECTEDVALUE (DAX function), 222, 228

self joins, 349

self-referencing relationship, 54

self-service BI, combining with enterprise BI

- basics, 67
- in DAX, 232
- in Power BI, 150-152
- in Power Query and M, 324
- in SQL, 414

semi-additive calculations, 46, 195

set operators

- basics, 11
- in DAX, 180-182
- in Power BI tables, 97
- in Power Query, 251
- in SQL, 341-344
 - EXCEPT operator, 344
 - INTERSECT operator, 343
 - UNION operator, 342

single table storage, 28

slowly changing dimensions (SCDs)

- basics
 - type 0: retaining original dimensions, 49
 - type 1: overwriting, 50
 - type 2: adding a new row, 51-52
 - type 3: adding new attributes, 52
 - type 4: adding mini-dimensions, 53
 - types 5, 6, 7: combining changes to dimensions, 53
- in data model, 49-53
- in DAX, 207-210
- in Power BI, 127-128
- in Power Query and M, 285

- in SQL, 387-395
 - type 0: retaining original dimensions, 388
 - type 1: overwriting, 389-392
 - type 2: adding a new row, 392-395

snowflake schemas, 34

source groups, 158

SQL (basics), xxviii

SQL (data model building), 361-397

- calculations, 376-379
- denormalizing, 375-376
- flags and indicators, 379-383
- hierarchies, 395-397
- normalizing, 362-374
 - creating a function, 370-371
 - creating a PROCEDURE, 371-374
 - creating a VIEW, 369
 - persisting into a table, 367-368
- role-playing dimensions, 385-387
- slowly changing dimensions, 387-395
 - type 0: retaining original dimensions, 388
 - type 1: overwriting, 389-392
 - type 2: adding a new row, 392-395
- time and date, 383-385

SQL (performance tuning), 417-430

- partitioning, 421-428
- pre-aggregating, 429
- storage modes, 417-421
 - compression, 420
 - function, 421
 - index, 418-419
 - stored procedure, 421
 - table, 418
 - view, 420

SQL (real-world examples), 399-415

- binning, 399-403
 - deriving lookup table from facts, 400-401
 - generating lookup table, 401
 - range tables, 402-403
- budget use case, 403
- combining self-service/enterprise BI, 414
- key-value pair tables, 408-413
- multi-language model, 404-408

SQL (relational data model basics), 335-360

- basic components, 336-341
 - foreign keys, 340-341
 - primary keys, 338

- relationships, 338
 - surrogate keys, 339
 - tables, 336-338
 - combining queries, 341-359
 - entity relationship diagrams, 356
 - extract, transform, load (ETL), 357-359
 - join path problems, 351-356
 - joins, 344-350
 - set operators, 341-344
 - data model, 335
 - SQL Server Management Studio (SSMS), xxxiii, 356
 - SQL Server, installing, xxxiii
 - star schema, xxv, 3, 34
 - (see also denormalizing)
 - storage modes
 - DAX, 235
 - DirectQuery, 154-160
 - dual mode, 169
 - hybrid tables, 170
 - import mode, 154
 - in SQL, 417-421
 - compression, 420
 - function, 421
 - index, 418-419
 - stored procedure, 421
 - table, 418
 - view, 420
 - live connection, 151
 - Power BI data model, 153-160
 - Power Query, 327
 - stored procedures, 421
 - SUM (DAX function), 195
 - SUMMARIZECOLUMNS (DAX function), 235
 - SUMX (DAX function), 195, 219
 - surrogate keys, 8
 - in Power BI, 94
 - in Power Query, 249-251
 - in SQL, 339
 - SWITCH (DAX function), 201
- ## T
- tables, 78-88
 - basics, 6
 - combining, 11-27, 97-100
 - entity relationship diagrams, 25-27, 100
 - join path problems, 20-25, 98
 - joins, 13-19, 97
 - in DAX, 176-179
 - in Power BI, 78-88
 - in Power Query, 244-247
 - in SQL, 336-338, 418
 - persisting into (SQL), 367-368
 - single table for storage, 28, 103-106
 - types of, 28, 101-103
 - Tabular Editor, 143
 - Tabular Translator, 144
 - time and date
 - basics, 47-48
 - with DAX, 203-205
 - with Power BI, 116-122
 - marking the date table, 121
 - turning off auto date/time, 116-121
 - with Power Query and M, 279-284
 - with SQL, 383-385
 - time-intelligence calculations, 198-200
 - transactional fact table, 35
 - TREATAS (DAX function), 179, 220-222, 224
 - TRUNCATE TABLE (SQL operator), 358
- ## U
- UI (user interface)
 - Power BI Desktop (standalone), 146
 - Power BI Desktop (Windows store), 147
 - Power BI report server, 148
 - Power BI service, 148
 - UNION (DAX function), 181
 - UNION (set operator), 251
 - UNION (SQL operator), 342
 - universal principal name (UPN), 225
 - USERCULTURE (DAX function), 223
 - USERRELATIONSHIP (DAX function), 124, 179, 206, 223
 - USERNAME (DAX function), 226
- ## V
- VALUE (DAX function), 229
 - VertiPaq engine, 41, 154
 - VIEW (SQL), 369
 - views, in SQL, 420
 - virtual relationships, 179
 - visual elements, in Power BI multi-language model, 140

About the Author

Markus Ehrenmueller-Jensen has been involved in data modeling for more than three decades. He teaches best practices at international conferences, webinars, and workshops and implements them for clients in various industries. Since 2006, he has specialized in building business intelligence and data warehousing solutions with Microsoft's Data Platform (the 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 the Fabric platform).

He is the founder of *Savory Data* and a professor of Databases and Information Systems at **HTL Leonding Technical College** and holds several Microsoft certifications. Markus is the (co)-author of several books. He has been repeatedly recognized as a Microsoft Data Platform MVP since 2017.

When Markus is not beating data into shape, he holds the beat behind the drum set in various musical formations. You can contact him via markus@savorydata.com, [LinkedIn](#), [X \(formerly Twitter\)](#), [Blue Sky](#), and [GitHub](#).

Colophon

The animal on the cover of *Data Modeling with Microsoft Power BI* is a fallow deer (*Dama dama*), the most common species of deer in the UK. Fallow deer graze on grasses, herbs, foliage, nuts, and fruit in deciduous woodland and grassland habitats. The deer can be various shades of brown but can be identified by a characteristic black-and-white tail with a white patch outlined in black on their hindquarters.

These animals typically live 8-10 years in the wild and stand about 3 feet tall from hoof to shoulder. Fallow deer are protected in the UK under the Deer Act of 1991.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on an antique line engraving from *Cassell's Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

The background of the entire page is a vibrant red-to-orange gradient. Overlaid on this are several large, semi-transparent, overlapping circles in various shades of red and orange, creating a dynamic, organic feel.

O'REILLY®

**Learn from experts.
Become one yourself.**

Books | Live online courses
Instant answers | Virtual events
Videos | Interactive learning

Get started at oreilly.com.