

SAMPLE

SECOND EDITION



OPTIMIZING DAX

Improving DAX performance in
Microsoft Power BI and Analysis Services

Alberto Ferrari
Marco Russo



Optimizing DAX

Improving DAX performance in
Microsoft Power BI and Analysis
Services

Second Edition

Alberto Ferrari and Marco Russo

Copyright © 2024 by Alberto Ferrari and Marco Russo

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. Microsoft and the trademarks listed at www.microsoft.com/en-us/legal/intellectualproperty/trademarks/usage/general are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, the publisher, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Publisher / Editorial Production: SQLBI Corp., Las Vegas, NV, Unites States

Revision: 2 (April 30, 2024)

Authors: Alberto Ferrari, Marco Russo

Copy Editor: Claire Costa

Cover Designer: Daniele Perilli

ISBN: 978-1-7353652-2-0

Library of Congress Control Number: 2024903659

Contents at a Glance

SECTION 1	CORE CONCEPTS	
CHAPTER 1	Introduction (S01.M01).....	3
CHAPTER 2	Introducing optimization with examples (S01.M02).....	11
CHAPTER 3	Introducing the Tabular query architecture (S01.M03).....	33
CHAPTER 4	Using the Power BI Desktop performance analyzer (S01.M04)	51
CHAPTER 5	Using DAX Studio (S01.M05).....	61
CHAPTER 6	Introducing query plans (S01.M06)	73
SECTION 2	THE FORMULA ENGINE	
CHAPTER 7	Understanding the DAX Formula Engine (S02.M01).....	83
CHAPTER 8	Understanding query plans (S02.M02).....	129
CHAPTER 9	Optimizing the formula engine (S02.M03).....	167
SECTION 3	VERTIPAQ	
CHAPTER 10	Understanding the VertiPaq engine (S03.M01).....	203
CHAPTER 11	Understanding VertiPaq relationships (S03.M02)	261
CHAPTER 12	Analyzing VertiPaq storage engine queries (S03.M03).....	287
CHAPTER 13	Optimizing common DAX constructs (S03.M04).....	337
CHAPTER 14	Moving and applying filters to tables (S03.M05).....	409
CHAPTER 15	Optimization examples for VertiPaq (S03.M06)	465
CHAPTER 16	Understanding security optimization (S03.M07).....	523
SECTION 4	DIRECTQUERY OVER SQL	
CHAPTER 17	Understanding DirectQuery over SQL (S04.M01).....	565
CHAPTER 18	Optimizing DirectQuery over SQL (S04.M02).....	597
CHAPTER 19	Optimization examples for DirectQuery (S04.M03)	673
SECTION 5	COMPOSITE MODELS	
CHAPTER 20	Understanding composite models (S05.M01).....	703
CHAPTER 21	Composite models optimization examples (S05.M02).....	733
CHAPTER 22	Understanding complex models (S05.M03)	769

Contents

SECTION 1	CORE CONCEPTS	
CHAPTER 1	Introduction (S01.M01).....	3
	Prerequisites.....	3
	Overview of the Tabular architecture.....	4
	Structure of the training.....	5
	Coding conventions.....	7
	Companion content.....	8
	<i>Software prerequisites</i>	8
	<i>Hardware prerequisites</i>	9
	<i>Download demos</i>	9
	<i>Sample code references</i>	10
CHAPTER 2	Introducing optimization with examples (S01.M02).....	11
	Optimizing DAX.....	11
	Optimizing the model.....	18
	Optimizing composite models.....	26
	Conclusions.....	32
CHAPTER 3	Introducing the Tabular query architecture (S01.M03).....	33
	Introducing the formula engine.....	33
	Introducing VertiPaq and DirectQuery architectures.....	35
	Introducing the VertiPaq storage engine.....	36
	Introducing the DirectQuery over SQL storage engine.....	38
	Introducing DirectQuery over AS.....	40
	Introducing data islands and cross-island query resolution.....	42
	Different types of models.....	47
	Conclusions.....	49
CHAPTER 4	Using the Power BI Desktop performance analyzer (S01.M04).....	51
	Running Performance Analyzer.....	51
	Understanding the numbers reported by Performance Analyzer.....	54
	Optimizing queries or measures?.....	56
	What can be optimized.....	57
	Saving performance data.....	60
	Conclusions.....	60
CHAPTER 5	Using DAX Studio (S01.M05).....	61
	Installing DAX Studio.....	61

	Introducing the metrics of a database	62
	Introducing All Queries	65
	Capturing Excel queries (MDX).....	66
	Introducing Load Performance Data.....	68
	Introducing Query Plan and Server Timings	70
	Conclusions	71
CHAPTER 6	Introducing query plans (S01.M06)	73
	Introducing the logical query plan	75
	Introducing the physical query plan	75
	Introducing storage engine queries.....	77
	Query plans in DirectQuery	78
	Conclusions	80
SECTION 2	THE FORMULA ENGINE	
CHAPTER 7	Understanding the DAX Formula Engine (S02.M01)	83
	Understanding datacaches	83
	Understanding materialization.....	90
	Understanding callbacks.....	96
	Formula engine with different storage engines	100
	Understanding vertical fusion.....	100
	Understanding horizontal fusion.....	105
	Examples of formula engine calculations.....	110
	<i>Sales of best products</i>	110
	<i>Top three colors</i>	115
	Measuring performance.....	118
	Gathering important timings from the query plan.....	119
	Analyzing query plans and timings	120
	<i>Year-to-date calculation of an additive measure</i>	120
	<i>Year-to-date calculation of a non-additive measure</i>	123
	Conclusions	127
CHAPTER 8	Understanding query plans (S02.M02)	129
	Query plan structure.....	129
	Query plan operator types.....	131
	<i>Properties of ScaLogOp</i>	131
	<i>Properties of RelLogOp</i>	132
	<i>Properties of LookupPhyOp</i>	133
	<i>Properties of IterPhyOp</i>	133
	<i>Properties of SpoolPhyOp</i>	134
	Interactions between the formula engine and the storage engine.....	134
	Common query plan operators.....	138
	Examples of query plans	140

	<i>Comparing SUM versus SUMX</i>	140
	<i>Comparing IF versus IF.EAGER</i>	143
	<i>Filtering with DAX versus using relationships</i>	152
	<i>Understanding SWITCH optimization</i>	157
	Conclusions	166
CHAPTER 9	Optimizing the formula engine (S02.M03).....	167
	Optimizing datacache use.....	167
	Sales of best products.....	173
	Running total of sales and ABC analysis.....	180
	Year-over-year customer growth as a percentage.....	189
	Conclusions	199
SECTION 3	VERTIPAQ	
CHAPTER 10	Understanding the VertiPq engine (S03.M01).....	203
	Using VertiPq Analyzer.....	203
	<i>Gathering vpax information with DAX Studio</i>	203
	<i>Analyzing a vpax with VertiPq Analyzer</i>	205
	Tabular data types.....	206
	Introduction to the VertiPq columnar database	207
	Understanding VertiPq compression.....	211
	<i>Understanding value encoding</i>	211
	<i>Understanding hash encoding</i>	213
	<i>Understanding run-length encoding</i>	214
	<i>Using VertiPq Analyzer to understand VertiPq compression</i>	216
	<i>Understanding re-encoding</i>	219
	Understanding segmentation and partitioning	221
	Understanding the importance of sorting	229
	Understanding VertiPq relationships.....	234
	Understanding attribute hierarchies	239
	Optimizing VertiPq models: examples	242
	<i>Sales amount versus quantity and net price</i>	242
	<i>Storing currency conversion data</i>	247
	<i>Date time versus date and time</i>	256
	Conclusions	260
CHAPTER 11	Understanding VertiPq relationships (S03.M02)	261
	Regular, unidirectional one-to-many relationships.....	262
	Regular, bidirectional one-to-many relationships.....	267
	Regular, one-to-one relationships	275
	Limited, many-to-many cardinality relationships.....	278
	Conclusions	285

CHAPTER 12	Analyzing VertiPaq storage engine queries (S03.M03)	287
	Analyzing simple xSQL queries	287
	Introducing basic VertiPaq functionalities	290
	Introducing batches	291
	Understanding internal and external SE queries	294
	Understanding distinct count in xSQL	297
	Understanding VertiPaq joins and filters	298
	<i>Introducing VertiPaq joins</i>	298
	<i>Introducing bitmap indexes</i>	299
	<i>Introducing reverse joins</i>	304
	Understanding VertiCalc and callbacks	308
	<i>Understanding CallbackDataID</i>	309
	<i>Understanding EncodeCallback</i>	314
	<i>Understanding LogAbsValueCallback</i>	319
	<i>Understanding RoundValueCallback</i>	321
	<i>Understanding MinMaxColumnPositionCallback</i>	324
	<i>Understanding Cond</i>	327
	Understanding the VertiPaq cache	329
	Choosing the correct data type for VertiPaq calculations	332
	Conclusions	335
CHAPTER 13	Optimizing common DAX constructs (S03.M04)	337
	Optimizing nested iterations	337
	Understanding the effect of context transition	349
	Different ways of performing a distinct count	353
	Optimizing LASTDATE calculations	365
	Avoid using SUMMARIZE and clustering	373
	Optimizing division by checking for zeroes	379
	Reducing the extent of the search by removing blanks	386
	Optimizing time intelligence calculations	395
	Distinct count over large cardinality columns	401
	Conclusions	408
CHAPTER 14	Moving and applying filters to tables (S03.M05)	409
	Different filters in CALCULATE	409
	<i>Analyzing single-column filters</i>	410
	<i>Analyzing multiple-column filters</i>	422
	<i>Analyzing filters over multiple tables</i>	427
	Understanding sparse or dense filters	428
	Filter columns, not tables	431
	Modeling many-to-many relationships	435
	<i>Testing the bidirectional model</i>	438
	<i>Testing the star model</i>	444

	<i>Testing the snake model</i>	447
	<i>Testing the advanced snake model</i>	457
	Conclusions	463
CHAPTER 15	Optimization examples for VertiPaq (S03.M06)	465
	Reducing nested iterations	465
	Optimizing complex filters in CALCULATE.....	472
	Optimizing Fusion Optimization.....	476
	Currency conversion	478
	Optimizing cumulative totals	486
	Average price variation of products over stores	494
	Optimizing the number of days with no sales.....	503
	Computing open orders.....	510
	Optimizing SWITCH and nested measures.....	516
	Conclusions	521
CHAPTER 16	Understanding security optimization (S03.M07).....	523
	Testing security conditions and their performance impact	523
	Understanding when and where security is enforced.....	524
	Understanding cached bitmap indexes and embedded filters.....	528
	Optimizing dynamic security.....	536
	Optimizing static security on the fact table.....	544
	Optimizing dynamic security on the fact table	552
	Conclusions	562
SECTION 4	DIRECTQUERY OVER SQL	
CHAPTER 17	Understanding DirectQuery over SQL (S04.M01).....	565
	Working with DirectQuery.....	565
	Reading SQL code in this book.....	566
	Reading the numbers in DAX Studio	568
	Callback operations.....	570
	Calculated tables.....	574
	Calculated columns.....	574
	How caching works in DirectQuery over SQL.....	576
	Understanding latency to send queries to the remote server	577
	Max number of rows in a data cache.....	577
	Different types of relationships.....	578
	<i>Regular one-to-many relationships</i>	578
	<i>Limited many-to-many relationships</i>	582
	<i>One-to-one relationships</i>	583
	DirectQuery over SQL max parallel queries.....	583
	Using different data islands	592

	Introducing aggregations and hybrid tables	596
	Conclusions	596
CHAPTER 18	Optimizing DirectQuery over SQL (S04.M02).....	597
	Building an SQL data model for Analysis Services	597
	<i>Designing indexes</i>	598
	<i>Using columnstore indexes</i>	601
	<i>Choosing column data types</i>	603
	<i>Do not use Power Query transformations</i>	607
	Optimizing relationships.....	607
	<i>Choosing the best data type for relationships</i>	608
	<i>Relying on referential integrity</i>	610
	<i>Using COMBINEVALUES to implement multi-column relationships</i>	615
	Using aggregations.....	623
	<i>Introducing aggregations</i>	623
	<i>Introducing VertiPaq aggregation and Dual storage mode</i>	631
	<i>Designing aggregations for simple calculations</i>	635
	<i>Designing aggregations for row-level calculations</i>	638
	<i>Designing aggregations for distinct counts</i>	642
	<i>Aggregations are not VertiPaq aliases of DirectQuery tables</i>	646
	<i>Manually activating aggregations in DAX</i>	652
	<i>Using automatic aggregations</i>	659
	Using hybrid tables	659
	<i>Introducing hybrid tables</i>	660
	<i>Reducing partition queries with DataCoverageDefinition</i>	668
	<i>Hybrid tables and distinct counts</i>	668
	<i>Creating hybrid tables with incremental refresh</i>	670
	Conclusions	672
CHAPTER 19	Optimization examples for DirectQuery (S04.M03).....	673
	Optimizing LASTDATE calculations.....	673
	Optimizing division by checking for zeroes	681
	Optimizing time intelligence calculations.....	687
	Computing distinct counts.....	695
	Conclusions	700
SECTION 5	COMPOSITE MODELS	
CHAPTER 20	Understanding composite models (S05.M01).....	703
	Introducing composite models.....	703
	Understanding wholesale and retail calculations	705
	Calculated tables.....	712
	Calculated columns.....	712
	Tracing remote queries	714

	Understanding relationships between tables	716
	Understanding special DAX functions for composite models	720
	<i>Understanding GROUPCROSSAPPLY and GROUPCROSSAPPLYTABLE</i>	721
	<i>Understanding DEPENDON</i>	723
	Splitting calculations between wholesale and retail	726
	Conclusions	732
CHAPTER 21	Composite models optimization examples (S05.M02).....	733
	Static segmentation	733
	Budget and time intelligence calculations	748
	Dynamic ABC analysis	759
CHAPTER 22	Understanding complex models (S05.M03)	769
	Understanding the role of the formula engine in complex models	769
	Calculated tables	773
	Calculated columns	773
	Relationships in complex models	780
	Using SQL Server features to avoid multiple data islands.....	790
	Using VertiPaq to snapshot expensive DirectQuery queries	794
	Conclusions	803

Section 1

CORE CONCEPTS

Introduction

This book is the written version of the Optimizing DAX video course (second edition) published by SQLBI. While the PDF version is available to video course students, we decided to publish a printed version for those who want a printed version or do not want to get the entire video course and prefer to study books.

Several years ago, we recorded and published the first edition of the Optimizing DAX video course. At the time, Tabular was much simpler than it is today, and the DAX optimization was described in a few chapters of “The Definitive Guide to DAX” book, including some details about the VertiPaq model. In the first edition of that training, we did not cover – on purpose – DirectQuery optimizations. Indeed, DirectQuery was not a real option at that time.

Over the years, Tabular has evolved, and its complexity is now much more significant. Microsoft introduced a more usable version of DirectQuery, released composite models, and improved the DAX engine by adding new optimization techniques. Moreover, as teachers, we increased our knowledge and experience about the engine in the same period. When it was time to create a new training version, we quickly discovered the amount of materials to teach was massive. Therefore, despite this being the second edition of the Optimizing DAX video course, it is – in reality – a brand new training (and book) about the state of the art of optimizing Tabular models.

An important note is that we wrote this content using Analysis Services 2022 and Power BI versions available in 2023. Different engine versions might show different behaviors, for the better or the worse. We will say this for the first time now and repeat it multiple times during the training: you must test any of the optimization techniques we teach on your models and the version of the engine you are currently using.

Prerequisites

The content of this book is advanced. As such, it certainly does not start from scratch. We take for granted that the reader has an excellent knowledge of several topics:

- **The DAX language.** In the training, we never teach DAX concepts. Here, the goal is how to produce efficient DAX code. We will write a lot of DAX code together, taking it for granted that you will quickly understand the different formulas. If you are unfamiliar with DAX, we strongly suggest the Mastering DAX video course, “The Definitive Guide to DAX” book, and practicing the DAX language in your daily job for at least one year. **Being proficient with DAX is an absolute requirement.** The book also uses query columns and query tables described in these articles:
 - **Introducing DEFINE COLUMN in DAX queries**
<https://www.sqlbi.com/articles/introducing-define-column-in-dax-queries/>

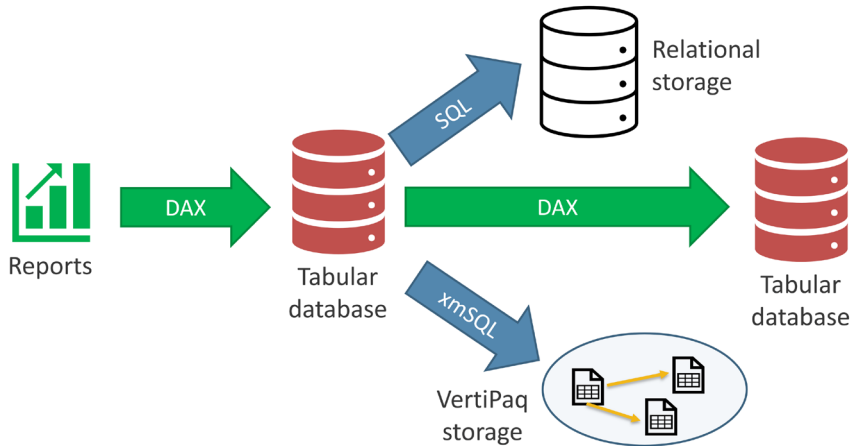
- **Introducing DEFINE TABLE in DAX queries**
<https://www.sqlbi.com/articles/introducing-define-table-in-dax-queries/>
- **The SQL language.** The modules about DirectQuery use quite some SQL code. You do not need to master all the details about SQL. Still, you must be able to read and understand SQL code quickly and how indexes, column stores, and other relational database technologies may affect your query's results and performance. We use Microsoft SQL Server in the book examples. You can apply the same concepts to other relational databases – even though you cannot use the same specific techniques described for Microsoft SQL Server.
- **The Tabular architecture.** Tabular is a complex engine that includes different technologies. We will use but do not explain many features of Tabular. Even though you can read and learn most of the content of this book without a deep understanding of the Tabular engine, we strongly suggest our students attend the Mastering Tabular video course first. Unfortunately, we do not have a corresponding book for that content.
- **DAX Studio.** DAX Studio is the primary tool used throughout the entire training. We introduce some basic concepts about DAX Studio but do not go in-depth on all the tool features. If you are unfamiliar with DAX Studio, we suggest you attend the free DAX Tools training first at <https://www.sqlbi.com/p/dax-tools-video-course/>.

Overview of the Tabular architecture

The architecture of a modern Tabular solution can be rather complex. To perform any optimization, we must understand many technical details about a Tabular solution. Indeed, when there are performance issues, users typically report that “the dashboard is slow” or “it takes too long to produce the report”. However, the report is only the tip of the iceberg of a performance issue hidden in any place of the architecture.

The following figure shows a very high-level overview of what happens when rendering a report:

Overview of a Tabular Architecture



The report sends a query to the Tabular model, which runs the query and might need to access an SQL database if the model uses DirectQuery, scan a VertiPaq storage if the model is in import mode, or connect to another Tabular model if we are using a composite model. Worse, it might be a mix of all these technologies together in a complex model.

The optimization techniques to use strongly depend on the architecture. We might face a simple DAX issue: poor code leads to bad performance. However, the same DAX code might work well with VertiPaq, and it might be a severe issue in a composite model. It might be the case that the code itself works fine, but the amount of materialization is excessive because the model uses DirectQuery.

In other words, providing guidelines about authoring good DAX code and building a performant model is nearly impossible unless we clearly understand all the pieces connected in a Tabular model.

Structure of the training

The book has five sections:

- **Section 1: Core concepts.** This introductory section starts with some optimization examples to provide an overview of the steps needed to optimize DAX code or Tabular models. Then, it covers the main topics about the Tabular architecture, includes information about the tools to use, and introduces the query plan concept.
- **Section 2: The formula engine.** This section is no longer an introduction. We go as deep as possible into the details of the formula engine, discover its operators, understand the datacaches, read the query plan, and then start optimizing code.
- **Section 3: VertiPaq.** Time for the main course. VertiPaq is the most critical section. We dive into the details of the VertiPaq engine through seven dense chapters packed with technical

information. The number of scenarios we optimize increases significantly, along with our knowledge.

- **Section 4: DirectQuery over SQL.** This section explores all the details about the DirectQuery over SQL storage engine. Please do not skip the previous sections of the book; they are required to understand how DirectQuery works.
- **Section 5: Composite models.** Composite models are a recent architectural addition to Tabular, and they prove to be more challenging because they share all the complexity of VertiPaq and DirectQuery.

A first read of the book from cover to cover reveals a long journey in DAX optimizations. Once you finish the book, you can use it as a reference to refresh your mind about specific topics.

We advise readers not to jump directly to the more advanced sessions until they are familiar with the previous topics. We build knowledge in steps and never repeat a topic multiple times. If you fast-forward to an advanced chapter, it is unlikely you will be able to appreciate several of the nuances.

Coding conventions

The book has many code samples in three languages: DAX, SQL, and xSQL. You will learn xSQL in this book, whereas you should already know DAX and SQL. Every language has a different style for code snippets that we present here to familiarize ourselves with.

Here is a sample code in DAX:

```
EVALUATE
SUMMARIZECOLUMNS (
    ROLLUPADDISSUBTOTAL (
        'Date'[Year],
        "IsGrandTotalRowTotal",
        ROLLUPGROUP ( 'Date'[Month], 'Date'[Month Number] ),
        "IsDM1Total"
    ),
    "Sales_Amount", 'Sales'[Sales Amount]
)
```

The following is SQL:

```
SELECT TOP (1000001) *
FROM (
    SELECT [t1_Year], SUM([a0]) AS [a0]
    FROM (
        SELECT [t1].[Year] AS [t1_Year],
            ([t3].[Quantity] * [t3].[Net Price]) AS [a0]
        FROM (
            [Data].[Sales] AS [t3]
            LEFT JOIN [dbo].[Date] AS [t1] ON ([t3].[Order Date] = [t1].[Date])
        )
        ) AS [t0]
    GROUP BY [t1_Year]
    ) AS [MainTable]
WHERE (NOT (([a0] IS NULL)))
```

And the last example is xSQL:

```
WITH
    $Expr0 := ( PFCAST ('Sales'[Quantity] AS INT) * PFCAST ('Sales'[Net Price] AS INT) )
SELECT
    'Product'[Brand],
    SUM ( @$Expr0 )
FROM 'Sales'
LEFT OUTER JOIN 'Product'
    ON 'Sales'[ProductKey]='Product'[ProductKey];
```

Companion content

All the examples included in the book can be downloaded and reproduced on your computer. You might see different absolute results because of differences in CPU, RAM, and software versions. However, most of the time, you will see the same relative differences between different optimization steps. Keep in mind the different baselines between your computers and ours. Because we used different hardware in different parts of the book, you will always find a reference to the CPU used in the initial comments of the DAX queries you can download.

You can download all the files following the link to **Companion content** on the <https://sql.bi/optdaxdemo> page.

In the following sections, we provide software and hardware prerequisites to use the demo files, how to download them, and how to find the demo files corresponding to each book chapter.

Software prerequisites

You should have the following software to practice using the book demos. If you want to know more about the optional tools, you can find more information in the Mastering Tabular video course from SQLBI at <https://www.sqlbi.com/p/mastering-tabular-video-course/>.

- **Power BI Desktop**
<https://powerbi.microsoft.com/downloads/>
- **DAX Studio**
<https://daxstudio.org/>
- **Excel** (optional)
<https://www.microsoft.com/microsoft-365/excel>
- **SQL Server 2022** (optional)
<https://www.microsoft.com/sql-server/>
- **SQL Server 2022 latest cumulative updates** (optional)
<https://www.microsoft.com/en-us/download/details.aspx?id=105013>
- **SQL Server Analysis Services Tabular** (optional)
<https://learn.microsoft.com/analysis-services/tabular-models/tabular-models-ssas>
- **SQL Server Management Studio** (optional)
<https://learn.microsoft.com/sql/ssms/download-sql-server-management-studio-ssms>
- **Tabular Editor 2 or 3** (optional)
<https://www.tabulareditor.com/>

Hardware prerequisites

To practice the demos with the largest PBIX demo file (Contoso 100M.pbix, 2GB), you need a computer with **at least 16GB of RAM**. If you do not have enough RAM, you can use the smaller Contoso 10M.pbix, which should run on any computer running Power BI Desktop, even though the metrics obtained can be significantly different from the ones shown in the book.

If you want to create the largest database for Analysis Services (Contoso 100M x 10), we suggest a computer with at least 64GB of RAM. This model is used in a few demos: you do not need to repeat those same demos, as you can see similar effects on smaller models, even though the results will be at a different order of magnitude.

Download demos

The demos are files we use in the book to illustrate how to measure performance and optimize code. You need Power BI Desktop and DAX Studio to repeat the same demos on your PC. For a few demos that require very large databases, you need SQL Server and Analysis Services: if you do not have enough RAM to install those databases, repeat the demos on the smaller Power BI Desktop files.

To practice the concepts learned, we suggest looking for patterns you have seen in each section by repeating the same analyses with the sample demo files provided and then on your models.

All the Contoso sample databases have the same structure; the only difference is their size. For example, Contoso 10K contains around 10,000 orders, while Contoso 1M contains around 1 million orders. Most of the demos run on Contoso 100M, and few run on a larger or smaller Contoso database: you find the reference to the Contoso version in the comments at the beginning of each DAX demo file.

The **PBIX files** contain the sample database for Power BI Desktop. You do not need to refresh these databases. Still, if you want, you should download the SQL Server backup (.bak files), restore them with the same database name (without the .bak extension) on a local SQL Server database, and create an alias named Demo that points to the SQL Server instance where you restored the bak files. Read the Creating database aliases on SQL Server 2022 article to create the Demo aliases on both 32-bit and 64-bit configurations of SQL Server: <https://sql.bi/creating-aliases/>.

The **Optimizing DAX - Demo files.ZIP** archive contains all the files used in the demos of the book. Download the file and unzip the content in a local folder on your computer. Most of the demos run on a Contoso sample database; only a limited number of demos have separated PBIX files that you find in the same module folder of the demo files.

The **BAK files** are SQL Server backups you can restore on SQL Server 2019 or 2022 to refresh the PBIX file and/or to populate corresponding Analysis Services databases. Read the Creating database aliases on SQL Server 2022 article to create the Demo aliases on both 32-bit and 64-bit configurations of SQL Server: <https://sql.bi/creating-aliases/>. The Contoso 100M.bak file is stored in two files, Contoso 100M.7z.001 and Contoso100M.7z.002: you can extract the Contoso 100M.bak using [7-Zip \(https://www.7-zip.org/\)](https://www.7-zip.org/).

The **AS BIM - Contoso 100M X 10.ZIP** archive contains the Tabular project (BIM file) to generate the

Analysis Services database "Contoso 100M x 10" used with the demos with the largest database. You need a large server with at least 64GB of RAM to process this database. However, you can use the Contoso 100M PBIX file to run the same demos executed on "Contoso 100M x 10" (see the reference database in the comments of the DAX code): you will see similar effects at a smaller order of magnitude.

Sample code references

At the beginning of each book chapter, there is a reference to the sample code. This reference is the corresponding section and module of the Optimizing DAX video course on SQLBI. For example, SAMPLE S02.M03 means that the sample code for the chapter is in the folder starting with "S02.M03" within the Demo folder. Such a reference also corresponds to Section 02 and Module 03 of the video course.

Introducing optimization with examples

The amount of knowledge required to optimize any measure is massive. The architecture of Tabular is rather complex, and we will dive into extremely technical details to appreciate the effect of any change.

It is easy to lose sight of the overall reason why certain specifics are important when diving into details. Therefore, if we were to follow an academic approach, and explain all the details before starting to use the concepts, the training would be extremely hard to follow, somewhat boring, and you would have to constantly go back to previous chapters to refresh your memory about topics that did not seem relevant the first time around.

We chose a different approach. This first set of introductory content shows the complete process of optimization through examples. We neither explain each step in full nor do we explain why we perform certain actions. The goal of this first set of optimizations is not that of learning how to optimize. Instead, we want to show the reasoning required to perform optimizations in different architectures.

Do not consider these first optimizations as best practices, and do not try to derive complete knowledge from these examples. These are just simple examples to show the overall process. Later on, we shall start to dive into the details. Then, hopefully, having seen the complete process will help you focus on the details without losing sight of the big picture.

During the process of optimizing the code, we also use several tools. Again, we do not provide a complete reference of those tools, because the goal is to show you when the tools are used rather than explain how to use them. We provide a much greater level of detail in the following chapters. In the entire description, we are deliberately concise. If the description seems cloudy to you, this is to be expected. Our promise is that you will be able to perform the same operations once you complete the training and have some more experience under your belt.

Optimizing DAX

We start by analyzing a simple measure that contains a DAX issue. The code is poorly written, and in the process of trying to optimize it, we will also make it worse before finally reaching the optimized version. The example is not relevant, despite being quite common. The critical detail to focus on is the entire optimization process.

It all starts with a report that users describe as slow. It contains a simple measure that computes sales for only transactions whose amount is greater than 200.00 USD.

Year	Sales Gt 200
2010	5,966,043,126.80
May	303,911,689.67
June	666,241,293.46
July	638,715,185.11
August	676,336,337.95
September	759,106,437.86
October	785,281,421.80
November	782,639,825.22
December	1,353,810,935.74
2011	10,691,471,516.63
2012	12,627,251,409.30
2013	20,443,916,137.00
2014	29,136,505,467.53
2015	23,549,371,252.73
2016	17,946,710,230.43
2017	30,342,774,851.17
Total	223,259,265,362.06

The code of the measure is the following:

```

-----
-- Measure in Sales table
-----
Sales Gt 200 =
SUMX (
    Sales,
    IF (
        Sales[Quantity] * Sales[Net Price] >= 200,
        Sales[Quantity] * Sales[Net Price]
    )
)

```

If you are a seasoned DAX developer, you know where the problem is right off the bat. Nonetheless, in this section we work on the full performance analysis of the measure.

The report is on a Power BI Desktop file, using Import mode. The *Sales* table contains 200M rows, spanning around 10 years of data. We use the Performance Analyzer tool in Power BI Desktop to retrieve the DAX query executed for the visual. Before analyzing the query, Performance Analyzer already provides us with some rather interesting numbers: the query took 1.2 seconds to run on a quite powerful server with 64 virtual cores.

Performance analyzer

Start recording Refresh visuals Stop

Clear Export

Name	Duration (ms)
Refreshed visual	-
Matrix	1210
DAX query	1159
Visual display	27
Other	24

Copy query

The code of the query is verbose, because it is automatically generated by Power BI:

```
// DAX Query
DEFINE
    VAR __DM3FilterTable =
        TREATAS ( { 2010 }, 'Date'[Year] )
    VAR __DS0Core =
        SUMMARIZECOLUMNS (
            ROLLUPADDISSUBTOTAL (
                'Date'[Year],
                "IsGrandTotalRowTotal",
                ROLLUPGROUP ( 'Date'[Month], 'Date'[Month Number] ),
                "IsDM1Total",
                NONVISUAL ( __DM3FilterTable )
            ),
            "Sales_Gt_200", 'Sales'[Sales Gt 200]
        )
    VAR __DS0PrimaryWindowed =
        TOPN (
            502,
            __DS0Core,
            [IsGrandTotalRowTotal], 0,
            'Date'[Year], 1,
            [IsDM1Total], 0,
            'Date'[Month Number], 1,
            'Date'[Month], 1
        )
EVALUATE
    __DS0PrimaryWindowed
ORDER BY
    [IsGrandTotalRowTotal] DESC,
    'Date'[Year],
    [IsDM1Total] DESC,
    'Date'[Month Number],
    'Date'[Month]
```

We want to simplify the query, to make it easier to understand. In the process of simplifying the query,

we need to pay attention not to get rid of the problem. The first thing to do is to execute the query in DAX Studio with Server Timings enabled to obtain the first baseline. Later on, we will check that the simplified query did not change the timings in such a way that the issue seems resolved. Here is the DAX Studio timings report.

Total	SE CPU	Line	Subclass	Duration	CPU	Par.	Rows	KB	Timeline
1,083 ms	50,360 ms x47.3	2	Scan	71	391	x5.5	225	4	
FE 19 ms 1.8%	SE 1,064 ms 98.2%	4	Scan	490	25,141	x51.3	14	1	
		6	Scan	503	24,828	x49.4	1	1	
SE Queries 3	SE Cache 0 0.0%								

Then, we simplify the query by removing the TOPN function, the final sorting, and other lines to make it shorter. We also add the definition of the measure, so we can change it later. Here is the shorter version we are going to work with:

```

DEFINE
    MEASURE Sales[Sales Gt 200] =
        SUMX (
            Sales,
            IF (
                Sales[Quantity] * Sales[Net Price] >= 200,
                Sales[Quantity] * Sales[Net Price]
            )
        )

EVALUATE
    SUMMARIZECOLUMNS (
        ROLLUPADDISSUBTOTAL (
            'Date'[Year],
            "IsGrandTotalRowTotal",
            ROLLUPGROUP ( 'Date'[Month], 'Date'[Month Number] ),
            "IsDM1Total"
        ),
        "Sales_Gt_200", 'Sales'[Sales Gt 200]
    )

```

The timings of this query are close to those of the previous one.

Total	SE CPU	Line	Subclass	Duration	CPU	Par.	Rows	KB	Timeline
1,433 ms	72,265 ms x51.0	2	Scan	493	24,203	x49.1	3,150	50	
FE 16 ms 1.1%	SE 1,417 ms 98.9%	4	Scan	457	23,828	x52.1	14	1	
		6	Scan	467	24,234	x51.9	1	1	


```

SET DC_KIND="AUTO";
WITH
  $Expr0 := [CallbackDataID ( IF (
    Sales[Quantity] * Sales[Net Price] >= 200,
    Sales[Quantity] * Sales[Net Price]
  ) )] ( PFDATAID ( 'Sales'[Quantity] ) , PFDATAID ( 'Sales'[Net Price] ) )
SELECT

```

Since the numbers are similar, we know that the problem is still there and we can start the optimization process. Despite the size of the *Sales* table being relatively large (200M is not huge, but it already is a significant number), the time required to compute the result seems excessive. The degree of parallelism is exceptional (on 64 cores, we obtained x51.0). Though the entire execution time is reported as storage engine CPU, we can clearly see a `CallbackDataID`, indicating that the formula engine is required to kick in to compute expressions that cannot be pushed down to the storage engine.

We know that a `CallbackDataID` is often a source of performance issues. It is nearly impossible to remove all `CallbackDataID`s from a query, but we can remove it in this situation. The problem is the `IF` statement inside the iteration carried on by `SUMX`, because the VertiPaq storage engine does not support conditional logic. We must rephrase the measure to avoid the `IF` statement; we replace it with a condition set by `CALCULATE` to rely on filtering rather than `IF`. A first (wrong) attempt in this direction is the following:

```

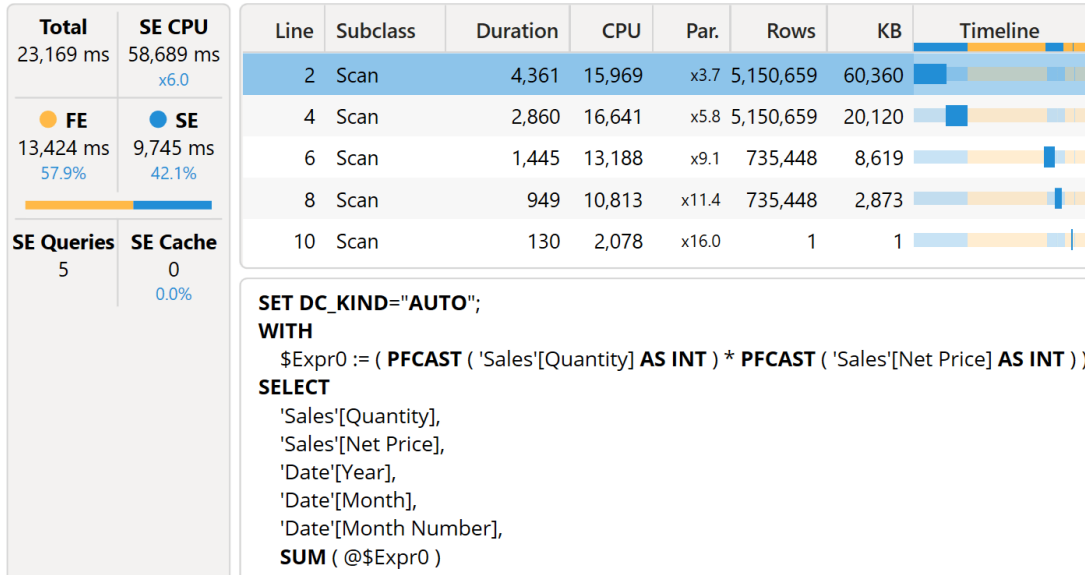
DEFINE
  MEASURE Sales[Sales Gt 200] =
    CALCULATE (
      SUMX ( Sales, Sales[Quantity] * Sales[Net Price] ),
      FILTER ( Sales, Sales[Quantity] * Sales[Net Price] >= 200 )
    )

EVALUATE
  SUMMARIZECOLUMNS (
    ROLLUPADDISSUBTOTAL (
      'Date'[Year],
      "IsGrandTotalRowTotal",
      ROLLUPGROUP ( 'Date'[Month], 'Date'[Month Number] ),
      "IsDM1Total"
    ),
    "Sales_Gt_200", 'Sales'[Sales Gt 200]
  )

```

The idea is to remove the requirement to compute `IF` by replacing it with a table filter computed by `FILTER` and applied by `CALCULATE`. It turns out to be an awful idea: the server timings are much worse

than before.



The storage engine CPU is a bit lower than the previous version of the measure, but the degree of parallelism is much lower this time (x6.0). Moreover, the formula engine executes a significant portion of code, making the overall performance much worse than the previous one. Overall, the execution time went from 1.5 to 23 seconds.

A deeper analysis of the xSQL queries shows that the storage engine is not actually computing the result. Although we have removed the CallbackDataID and replaced it with an xSQL filter, the VertiPaq engine retrieves way too much data. The following is the first xSQL query in the previous screenshot:

```

WITH
  $Expr0 := ( PFCAST ( 'Sales'[Quantity] AS INT ) * PFCAST ( 'Sales'[Net Price] AS INT ) )
SELECT
  'Sales'[Quantity],
  'Sales'[Net Price],
  'Date'[Year],
  'Date'[Month],
  'Date'[Month Number],
  SUM ( @$Expr0 )
FROM 'Sales'
LEFT OUTER JOIN 'Date'
  ON 'Sales'[Order Date]='Date'[Date]
WHERE
  ( COALESCE ( ( PFCAST ( 'Sales'[Quantity] AS INT ) * PFCAST ( 'Sales'[Net Price] AS INT ) ) ) >= COALESCE ( 2000000 ) ) ;

```

Even though the query groups by *Date[Year]*, *Date[Year Month]* (and *Date[Year Month Number]* because of the sort-by-column property), the xSQL query also groups by *Sales[Quantity]* and *Sales[Net*

Price]. The reduced degree of parallelism is mainly due to the large size of the datacaches returned by the storage engine to the formula engine. Moreover, the formula engine must carry out the calculation because the VertiPaq result in the datacache does not contain the result. Hence the exaggerated time required from the formula engine. The problem is the large datacache size, resulting in extreme materialization. The technique in itself is smart. The problem is the way we expressed the query.

Indeed, we used a filter over *Sales* as a filter argument in *CALCULATE*. A table filter is a very bad practice that newbies oftentimes use. A seasoned DAX developer knows that a filter in *CALCULATE* should work on the minimum number of columns required to obtain its effect. The filter over *Sales* in *CALCULATE* is for sure the issue in this measure. Therefore, we move forward and replace it with a filter over the only two columns required to apply their effect, which are *Sales[Quantity]* and *Sales[Net Price]*:

```
DEFINE
    MEASURE Sales[Sales Gt 200] =
        CALCULATE (
            SUMX ( Sales, Sales[Quantity] * Sales[Net Price] ),
            FILTER (
                ALL ( Sales[Quantity], Sales[Net Price] ),
                Sales[Quantity] * Sales[Net Price] >= 200
            )
        )

EVALUATE
    SUMMARIZECOLUMNS (
        ROLLUPADDISSUBTOTAL (
            'Date'[Year],
            "IsGrandTotalRowTotal",
            ROLLUPGROUP ( 'Date'[Month], 'Date'[Month Number] ),
            "IsDM1Total"
        ),
        "Sales_Gt_200", 'Sales'[Sales Gt 200]
    )
```

The result is exactly what we were searching for.

Total 197 ms	SE CPU 4,984 ms x27.5	Line	Subclass	Duration	CPU	Par.	Rows	KB	Timeline
FE 16 ms 8.1%	SE 181 ms 92.9%	2	Scan	183	4,984	x27.2	3,150	50	
SE Queries 1	SE Cache 0 0.0%	<pre> SET DC_KIND="AUTO"; WITH \$Expr0 := (PFCAST ('Sales'[Quantity] AS INT) * PFCAST ('Sales'[Net Price] AS INT)) SELECT 'Date'[Year], 'Date'[Month], 'Date'[Month Number], SUM (@\$Expr0) FROM 'Sales' LEFT OUTER JOIN 'Date' ON 'Sales'[Order Date]='Date'[Date] </pre>							

All the indicators are just perfect. The storage engine CPU is massively reduced, the degree of parallelism is back to being exceptional, there is virtually no formula engine involved in the query and no CallbackDataIds anywhere. Materialization is reduced from 5 million rows to only 3,150 rows. Overall, the execution went from 1.5 seconds to a bit less than 200 hundreds milliseconds – which given the size of the *Sales* table, was expected.

Job done; the code is now good enough to go into production and replace the previous measure. In order to complete our task we had to leverage on our DAX knowledge to rephrase the code, we had to discover what is being computed by the formula engine and the storage engine, we had to check that the degree of parallelism was the one expected, and we used the size of the data caches as an indicator of the materialization level. We used Performance Analyzer and DAX Studio to obtain our goal. We will learn all these details. For now, let us move on to the next example.

Optimizing the model

In the previous example, we optimized a piece of DAX code: The goal was to reduce the execution time of a query. We started from a measure because it is likely to be the most common optimization requirement. Nonetheless, other types of optimization are equally important. For example, reducing the size of a model improves both the execution time and the memory usage of a data model.

As in the previous example, we are not interested in providing detailed information about our considerations to optimize a model. Rather, we want to share the steps of optimizing a model by showing the reasoning behind a choice. In this example, we must choose between a calculated column and a calculation at query time.

The *Sales* table contains two columns – *Sales[Quantity]* and *Sales[Net Price]* – used to compute the sales amount by summing the quantity multiplied by the net price. This calculation is widely used in reports, along with other columns like *Quantity*, *Net Price*, and *Unit Cost*.

This is the code of the *Sales Amount* measure:

```

-----
-- Measure in Sales table
-----
Sales Amount = SUMX ( Sales, Sales[Quantity] * Sales[Net Price] )

```

The question is rather simple. Is it better to perform the multiplication at query time – using an iterator like we are doing in *Sales Amount* – or is it better to create a calculated column containing the multiplication result and then sum the calculated column at query time?

Creating a calculated column comes with several consequences:

- The model size grows because the calculated column is stored in RAM;
- The model processing time increases because the calculated column is computed sequentially at process time.

The price to pay for a calculated column is a larger model that takes more time to process. At the same time, we expect that computing the value in advance will reduce the execution time of queries. To make an educated decision, we evaluate the different impacts.

Let us start by creating the calculated column:

```

-----
-- Calculated column in Sales table
-----
Line Amount (USD) = Sales[Quantity] * Sales[Net Price]

```

We must evaluate the impact on both process time and size. Before measuring the time required to compute this calculated column, we note that the column will be used only in DAX calculations: therefore, we do not need to store its hierarchy. The hierarchy would consume processing time and space in memory, so we clear the *AvailableInMDX* property of the column.

Once we have created the column in the model, we process the model, triggering this way the calculation of only this column. We cannot easily measure the time required to compute only this new column. Nonetheless, because right after its creation the column is the only unprocessed entity in the entire model, measuring the default process of the model provides a good approximation of the actual cost.

Before starting the process default, we open SQL Server Profiler and trace the Progress Report events. Then, we run a refresh by using the following TMSL script in SQL Server Management Studio:


```

{
  "refresh": {
    "type": "automatic",
    "objects": [
      {
        "database": "Contoso 100M x 10"
      }
    ]
  }
}

```

Because we monitor the process and the profiler gathers all progress report events, we should avoid using the server for anything else until the process finishes. The model is quite large; the *Sales* table contains 1.4B rows, so we do not need to measure timings in an extremely precise way.

Once the processing is finished, we can look at the events to discover the time required to compute the column.

EventClass	TextData	CurrentTime
Progress Report Begin		2023-03-13 17:18:40.000
Progress Report End		2023-03-13 17:18:40.000
Progress Report Begin	Started sequence point algorithm.	2023-03-13 17:18:40.000
Progress Report Begin	Started processing calculated column 'Line Amount (USD)' of table 'Sales'.	2023-03-13 17:18:40.000
Progress Report End	Finished processing calculated column 'Line Amount (USD)' of table 'Sales'.	2023-03-13 17:20:22.000
Progress Report Begin	Started processing hierarchy 'Line Amount (USD)' of table 'Sales'.	2023-03-13 17:20:22.000
Progress Report End	Finished processing hierarchy 'Line Amount (USD)' of table 'Sales'.	2023-03-13 17:20:22.000
Progress Report End	Finished sequence point algorithm.	2023-03-13 17:20:22.000
Progress Report Begin		2023-03-13 17:20:22.000
Progress Report Begin	Started Phase 1 of the Commit operation in a Tabular transaction.	2023-03-13 17:20:22.000
Progress Report End	Finished Phase 1 of the Commit operation in a Tabular transaction.	2023-03-13 17:20:23.000
Progress Report Begin	Started Phase 2 of the Commit operation in a Tabular transaction.	2023-03-13 17:20:23.000
Progress Report End	Finished Phase 2 of the Commit operation in a Tabular transaction.	2023-03-13 17:20:23.000
Progress Report End		2023-03-13 17:20:23.000

The processing started at 17:18:40 and finished at 17:20:22, meaning that the calculated column took 1 minutes and 42 seconds to compute. It is also worth noting that the server used a single core during the entire calculation. This is a known limitation of AS, which computes calculated columns sequentially. The time spent to compute a calculated column cannot be reduced by increasing the number of cores. Nonetheless, we also know that if the result is good we can compute this column as a calculated column in the view that feeds the table – so that instead of being a calculated column, the *Line Amount* column becomes a regular imported column.

We perform the tests with a calculated column because we do not want to process 1.4B rows every time we run a test. The results with a calculated column are slightly imprecise, but the time required to gather them is so tiny that it is worth proceeding this way.

We can now use the VertiPaq Analyzer information to assess the size of the column.


Name	Cardinality	Total Size ↓	Data	Dictionary	Hier Size
Sales	1,405,546,659	27,321,446,296	22,168,420,224	5,126,682,000	16,372,000
Order Number	94,138,244	10,696,887,352	5,617,629,840	5,079,257,512	0
CustomerKey	1,868,002	3,804,096,344	3,748,125,560	41,026,752	14,944,032
Line Amount (USD)	172,618	3,466,022,136	3,459,708,952	4,932,224	1,380,960
Net Price	24,754	2,811,721,664	2,811,094,736	626,928	0
ProductKey	2,517	2,248,971,852	2,248,875,792	75,916	20,144
Unit Cost	1,956	1,920,667,096	1,920,625,464	41,632	0
Unit Price	1,761	1,920,661,660	1,920,621,304	40,356	0
Line Number	20	385,448,264	385,446,848	1,416	0
StoreKey	74	38,664,984	38,661,720	2,656	608
Delivery Date	3,389	8,793,568	8,615,664	177,904	0
Quantity	10	7,827,232	7,825,856	1,376	0
Order Date	3,281	924,296	722,360	175,680	26,256
Exchange Rate	5,071	765,328	460,944	304,384	0
Currency Code	5	22,312	5,184	17,128	0

The entire *Sales* table uses 27GB of RAM, with the *Line Amount (USD)* column alone using 3.4GB. In other words, the calculated column alone uses around 14% of the total size of the database. The impact is quite significant in terms of RAM.

Now that the column is in place, we can measure the benefits – if any – in terms of performance. Because we are interested in an overall measure, we do not slice by any column and just compute the sales amount with either the new calculated column or the iteration. We author the following query and compute first the Calc Column row alone, and then the Measure row, alone again:

```
EVALUATE
SUMMARIZECOLUMNS (
    "Calc column", SUM ( Sales[Line Amount (USD)] )
)
```


Here is the result with the calculated column.

Total 90 ms	SE CPU 1,781 ms x19.8	Line	Subclass	Duration	CPU	Par.	Rows	KB	Timeline
● FE 0 ms 0.0%	● SE 90 ms 100.0%	2	Scan	90	1,781	x19.8	1	1	
SE Queries 1	SE Cache 0 0.0%								

The important number is the SE CPU, which is not dependent on the number of cores. Indeed, we use a server with a very large number of cores, reaching a very high degree of parallelism. SE CPU is a better indicator of the raw power consumed to produce the result when evaluating performance.

Next, the result using the measure, therefore performing the multiplication for every row:

```
EVALUATE
SUMMARIZECOLUMNS (
    "Measure", SUMX ( Sales, Sales[Quantity] * Sales[Net Price] )
)
```

Total 210 ms	SE CPU 4,016 ms x19.1	Line	Subclass	Duration	CPU	Par.	Rows	KB	Timeline
● FE 0 ms 0.0%	● SE 210 ms 100.0%	2	Scan	210	4,016	x19.1	1	1	
SE Queries 1	SE Cache 0 0.0%								

As expected, the calculated column is beneficial to performance. The calculated column takes 1,781 milliseconds to aggregate its values, whereas the multiplication at query time produces the same result using 4,016 milliseconds of SE CPU.

The question is whether the benefit is worth the increase in size. Before making up our mind, we want to look at further considerations. The test focuses strictly on the *Line Amount* calculated column. Nonetheless, we have other calculations following the same pattern: cost and margin. Therefore, we create two more calculated columns and measures:

```

-----
-- Calculated column in Sales table
-----
Line Cost (USD) = Sales[Quantity] * Sales[Unit Cost]

-----
-- Calculated column in Sales table
-----
Line Margin (USD) = Sales[Quantity] * ( Sales[Net Price] - Sales[Unit Cost] )

-----
-- Measure in Sales table
-----
Cost Amount = SUMX ( Sales, Sales[Quantity] * Sales[Unit Cost] )

-----
-- Measure in Sales table
-----
Margin Amount = SUMX ( Sales, Sales[Quantity] * ( Sales[Net Price] - Sales[Unit Cost] ) )

```

Then, we repeat the measurement process in a more complex query that uses the three columns and measures. First, we run this query to measure the time required to compute the values in the calculated columns:

```

EVALUATE
SUMMARIZECOLUMNS (
    "Sales Amount", SUM ( Sales[Line Amount (USD)] ),
    "Cost Amount", SUM ( Sales[Line Cost (USD)] ),
    "Margin Amount", SUM ( Sales[Line Margin (USD)] )
)

```

The timing is in line with what we expect; the time required to compute three measures is around three times the time required to compute one measure.

Total 257 ms	SE CPU 4,703 ms x18.3	Line	Subclass	Duration	CPU	Par.	Rows	KB	Timeline
		2	Scan	257	4,703	x18.3	1	1	
● FE 0 ms 0.0%	● SE 257 ms 100.0%	SET DC_KIND="AUTO"; SELECT SUM ('Sales'[Line Margin USD]), SUM ('Sales'[Line Amount USD]), SUM ('Sales'[Line Cost USD]) FROM 'Sales';							
SE Queries 1	SE Cache 0 0.0%								

Then, we execute the following query to evaluate the time required to compute the values at query time:

```

EVALUATE
SUMMARIZECOLUMNS (
    "Sales Amount", [Sales Amount],
    "Cost Amount", [Cost Amount],
    "Margin Amount", [Margin Amount]
)

```

This time, the results are worse than expected, because of the larger time in the SE CPU.

Total 783 ms	SE CPU 15,953 ms x20.5	Line	Subclass	Duration	CPU	Par.	Rows	KB	Timeline
		2	Scan	780	15,953	x20.5	1	1	
FE 3 ms 0.4%	SE 780 ms 99.6%	SET DC_KIND="AUTO"; WITH \$Expr0 := (PFCAST ('Sales'[Quantity] AS INT) * (PFCAST ('Sales'[Net Price] AS INT) \$Expr1 := (PFCAST ('Sales'[Quantity] AS INT) * PFCAST ('Sales'[Net Price] AS INT)) \$Expr2 := (PFCAST ('Sales'[Quantity] AS INT) * PFCAST ('Sales'[Unit Cost] AS INT)) SELECT SUM (@\$Expr0), SUM (@\$Expr1), SUM (@\$Expr2) FROM 'Sales';							
SE Queries 1	SE Cache 0 0.0%								

Indeed, gathering three columns using calculated columns increased the time required by a factor of around three. On the other hand, the same results computed by using measures are slower: 16 seconds against 4 seconds – four times more expensive. The reason is that *Line Margin* requires multiple columns in the same expression: *Quantity*, *Net Price*, and *Unit Cost*. Increasing the expression's complexity makes the solution with the calculated column preferable.

Nonetheless, we should also evaluate the increase in terms of RAM. Here is the VertiPaq Analyzer report with the three calculated columns.

Name	Cardinality	Total Size ↓	Data	Dictionary	Hier Size
Sales	1,405,546,659	33,588,398,696	28,427,946,...	5,132,227,3...	18,252,448
Order Number	94,138,244	10,696,887,352	5,617,629,840	5,079,257,512	0
CustomerKey	1,868,002	3,804,096,344	3,748,125,560	41,026,752	14,944,032
Line Amount (USD)	172,618	3,466,022,136	3,459,708,952	4,932,224	1,380,960
Line Margin (USD)	219,718	3,455,498,696	3,448,524,152	5,216,784	1,757,760
Net Price	24,754	2,811,721,664	2,811,094,736	626,928	0
Line Cost (USD)	15,334	2,811,453,704	2,811,002,488	328,528	122,688
ProductKey	2,517	2,248,971,852	2,248,875,792	75,916	20,144
Unit Cost	1,956	1,920,667,096	1,920,625,464	41,632	0
Unit Price	1,761	1,920,661,660	1,920,621,304	40,356	0
Line Number	20	385,448,264	385,446,848	1,416	0
StoreKey	74	38,664,984	38,661,720	2,656	608
Delivery Date	3,389	8,793,568	8,615,664	177,904	0
Quantity	10	7,827,232	7,825,856	1,376	0
Order Date	3,281	924,296	722,360	175,680	26,256
Exchange Rate	5,071	765,328	460,944	304,384	0
Currency Code	5	22,312	5,184	17,128	0

The three columns use 9.6GB of RAM, massively increasing the model size.

Though it is undoubtedly true that a calculated column brings good benefits to the model performance, at the same time, the price in terms of RAM consumption seems excessive. Creating one calculated column for each calculation would make the model too large.

Here is where a conclusion is hard to make without considering further details. If this were a general-purpose model, it would not make much sense to consolidate all the possible calculations in calculated columns, because the price in terms of space used would definitely be too large. On the other hand, if you have a complex calculation that involves multiple columns and intricate logic, and this calculation is at the core of most of your reports, then a calculated column to consolidate those results would make sense.

In the particular case of Contoso, with *Sales Amount*, *Cost Amount*, and *Margin Amount*, it is likely better to avoid the calculated columns to save memory, thus paying the price of working with slower queries – even though in smaller models the difference might not be measurable.

As you have seen, a simple decision about whether to compute a calculated column or not requires quite a few considerations and measurements. Besides, just knowing the details of why we made a choice already has a lot of value. We did not choose randomly. We performed measurements and made decisions based on solid numbers. If at any point we need to change our mind, we already know the consequences of making any choice.

Optimizing composite models

The first optimization example we saw in this chapter was related to a simple DAX optimization. For the sake of simplicity, we used an example using a regular import model, therefore handling only xSQL queries to the VertiPaq engine. Nonetheless, it is important to note that optimizing DAX requires a deep understanding of the entire model architecture.

As an example, we see how the composite model feature affects the process of DAX optimization. We use as an example the dynamic segmentation pattern. We will experience bad performance: the reason is not the DAX code, but rather the interaction between the local and remote engines.

This example aims to demonstrate that optimizing DAX requires more than just DAX knowledge. The same DAX code performs very differently if executed in a regular VertiPaq model, or in a composite model. In order to optimize DAX for a composite model, you need to have a profound understanding of the architecture and how the queries are split between the remote and the local servers.

We use a variation of the pattern available at [Dynamic segmentation – DAX Patterns \(https://www.daxpatterns.com/dynamic-segmentation/\)](https://www.daxpatterns.com/dynamic-segmentation/) We implement dynamic clustering on a model hosted in the Power BI Service by extending it with a composite model. Based on a configuration table, we want to dynamically cluster the customers by *Sales Amount*. The following is the configuration table created in the composite model.

Segment	MinValue	MaxValue
Low Sales	0	1,000
Medium Sales	1,000	10,000
High Sales	10,000	999,999,999

We create a measure using a slightly simplified version of the code published on www.daxpatterns.com:

```

-----
-- Measure in Sales table
-----
Customer in segment :=
SUMX (
    'Sales Segment',
    VAR MinSale = 'Sales Segment'[MinValue]
    VAR MaxSale = 'Sales Segment'[MaxValue]
    VAR CustInSeg =
        FILTER (
            Customer,
            VAR CustSales = [Sales Amount]
            RETURN
                CustSales > MinSale && CustSales <= MaxSale
        )
    RETURN
        COUNTROWS ( CustInSeg )
)

```

The measure mixes the *Sales Segment* table stored in the local VertiPaq model with the *Customer* table stored in the remote model. It works just fine: you can generate the following report showing the number of customers in each segment, sliced by year.

Segment	2017	2018	2019	2020	Total
High Sales	45	56	38	7	152
Low Sales	459	1,071	854	213	2,478
Medium Sales	656	1,236	903	205	2,955
Total	1,160	2,363	1,795	425	5,585

Although the numbers are correct, there is a performance issue. The measure would work very well in a regular VertiPaq model, but it is extremely slow in a composite model. The query that populates the matrix (somewhat simplified here) is the following:

```

--
-- Query executed on the local model
--
EVALUATE
SUMMARIZECOLUMNS (
    ROLLUPADDISSUBTOTAL ( 'Sales Segment'[Segment], "IsGrandTotalRowTotal" ),
    ROLLUPADDISSUBTOTAL ( 'Date'[Year], "IsGrandTotalColumnTotal" ),
    "Customer_in_segment", 'Sales Segment'[Customer in segment]
)

```

Looking at the server timings, we discover that this simple query runs in around three seconds on a rather small model with a few thousand sales:

Total	SE CPU	Line	Subclass	Duration	CPU	Par.	Rows	KB	Timeline	Query
2,721 ms	4,486 ms x1.7	1	DAX	554	554	x1.0				EVALUATE S
FE SE		2	DAX	1,893	1,891	x1.0				DEFINE VAR
123 ms	2,598 ms (2,598 ms)	3	DAX	1,987	1,987	x1.0				DEFINE VAR
4.5%	95.5%	5	Scan	1	0		3	1		SELECT 'Sale
SE Queries	SE Cache	6	DAX	54	54	x1.0				EVALUATE S
10	0	8	Scan	0	0		3	1		SELECT 'Sale
	0.0%	10	Scan	0	0		3	1		SELECT 'Sale

Further analysis shows that the complexity is hidden in the storage engine DAX queries sent to the remote server. If we analyze those queries, we find a first query that is already suspicious:

```
--
-- DAX DirectQuery query executed on the remote model
--
EVALUATE
SELECTCOLUMNS (
    'Customer',
    "__RN", blank(),
    "'Customer'[CustomerKey]", 'Customer'[CustomerKey],
    "'Customer'[Gender]", 'Customer'[Gender],
    "'Customer'[Name]", 'Customer'[Name],
    "'Customer'[Address]", 'Customer'[Address],
    "'Customer'[City]", 'Customer'[City],
    "'Customer'[State Code]", 'Customer'[State Code],
    "'Customer'[State]", 'Customer'[State],
    "'Customer'[Zip Code]", 'Customer'[Zip Code],
    "'Customer'[Country Code]", 'Customer'[Country Code],
    "'Customer'[Country]", 'Customer'[Country],
    "'Customer'[Continent]", 'Customer'[Continent],
    "'Customer'[Birthday]", 'Customer'[Birthday],
    "'Customer'[Age]", 'Customer'[Age]
)
```

This query retrieves all the columns of the *Customer* table. In our example, there are just a few thousand customers. In a more realistic scenario, you could have millions of customers. Therefore, the query is simple, but the datacache returned is potentially huge. Besides, it seems like neither the measure nor the report require any of those columns.

The next two queries are even worse, and they provide an explanation about why the previous query retrieved the entire *Customer* table. The first query retrieves the sales amount for each customer and year, using as a filter the entire customer table. In other words, the customer table that the local DAX engine has retrieved from the remote model is used as a filter in a subsequent query sent to the remote engine:

```
--
-- DAX DirectQuery query executed on the remote model
```

```

--
DEFINE
VAR _Var0 =
VALUES ( 'Date'[Year] )
VAR _Var1 =
SUMMARIZE (
    'Customer',
    'Customer'[CustomerKey],
    'Customer'[Gender],
    'Customer'[Name],
    'Customer'[Address],
    'Customer'[City],
    'Customer'[State Code],
    'Customer'[State],
    'Customer'[Zip Code],
    'Customer'[Country Code],
    'Customer'[Country],
    'Customer'[Continent],
    'Customer'[Birthday],
    'Customer'[Age]
)
VAR _Var2 = {
( 1212508, "Male", "David Puente", "189 Koontz Lane", "Los Angeles", "CA",
"California", "90017", "US", "United States", "North America", DT"1992-1-4", 29 ),
( 1200226, "Male", "William Gaughan", "4384 Euclid Avenue", "Guadalupe", "CA",
"California", "93434", "US", "United States", "North America", DT"1942-6-26", 78 ),
--
--      Several thousands of rows here, one for each customer
--
( 1200334, "Male", "Micheal Boyers", "655 Carriage Court", "Los Angeles", "CA",
"California", "90017", "US", "United States", "North America", DT"1989-9-30", 31 ),
( 1201225, "Male", "John Lally", "3886 Kerry Way", "Irvine", "CA",
"California", "92614", "US", "United States", "North America", DT"1996-2-6", 24 ),
( 395073, "Male", "Michael Vandermark", "386 Dufferin Street", "Toronto", "ON",
"Ontario", "M6H 4B6", "CA", "Canada", "North America", DT"1983-1-3", 38 )
}
EVALUATE
GROUPCROSSAPPLYTABLE (
    'Date'[Year],
    _Var0,
    _Var1,
    "L1",
    GROUPCROSSAPPLY (
        'Customer'[CustomerKey],
        'Customer'[Gender],
        'Customer'[Name],
        'Customer'[Address],
        'Customer'[City],
        'Customer'[State Code],
        'Customer'[State],
        'Customer'[Zip Code],
        'Customer'[Country Code],
        'Customer'[Country],
        'Customer'[Continent],
        'Customer'[Birthday],

```

```
'Customer'[Age],
KEEPFILTERS (
  TREATAS (
    _Var2,
    'Customer'[CustomerKey],
    'Customer'[Gender],
    'Customer'[Name],
    'Customer'[Address],
    'Customer'[City],
    'Customer'[State Code],
    'Customer'[State],
    'Customer'[Zip Code],
    'Customer'[Country Code],
    'Customer'[Country],
    'Customer'[Continent],
    'Customer'[Birthday],
    'Customer'[Age]
  )
),
"__Agg0", [Sales Amount]
)
```

Not only is this query huge, but so is its resulting data cache. Indeed, the query result is – again – the entire *Customer* table along with the *Sales Amount* result. Therefore, this is a large query with a large result... Definitely an issue.

A third query is very similar to the previous one, where the only noticeable difference is that the year is no longer part of the group by section. Still, the query includes the entire *Customer* table, and the resulting datacache contains the entire *Customer* table.

In other words, the entire *Customer* table is passed back and forth between the local and the remote servers multiple times. The number of customers in each segment is later computed in the formula engine based on the data retrieved from the remote server.

Now that the reason why the query is slow is somewhat clear, it is time to fix the problem. One of the few golden rules of DAX is to never use a table to apply a filter. A filter over a table requires scanning the entire table. The VertiPaaS engine contains several optimizations and patterns that reduce the number of columns scanned. However, the DirectQuery over AS engine has a reduced set of optimizations. Therefore, Tabular ends up scanning the entire table, just because this is what we required in the DAX code.

More specifically, the problem is the reference to *Customer* in the FILTER part of the measure. It turns out that we do not need to count the rows in the *Customer* table; it is enough to retrieve only the *Customer[CustomerKey]* column to obtain a semantically equivalent measure:

```

-----
-- Measure in Sales table
-----
Customer in segment =
SUMX (
    'Sales Segment',
    VAR MinSale = 'Sales Segment'[MinValue]
    VAR MaxSale = 'Sales Segment'[MaxValue]
    VAR CustInSeg =
        FILTER (
            DISTINCT ( Customer[CustomerKey] ),
            VAR CustSales = [Sales Amount]
            RETURN
                CustSales > MinSale && CustSales <= MaxSale
        )
    RETURN
        COUNTROWS ( CustInSeg )
)

```

By operating this small change in the formula, we obtain the very same result, but the DAX queries sent to the remote engine are much simpler, hence the timings.

Total	SE CPU	Line	Subclass	Duration	CPU	Par.	Rows	KB	Timeline	Query
603 ms	661 ms x1.2	1	DAX	102	102	x1.0				DEFINE VA
FE ⚠	SE	2	DAX	109	109	x1.0				DEFINE VA
37 ms	566 ms (566 ms)	3	DAX	93	93	x1.0				EVALUATE
6.1%	93.9%	5	Scan	3	0		3	1		SELECT 'Sa
		6	DAX	53	52	x1.0				EVALUATE
SE Queries	SE Cache	7	DAX	95	95	x1.0				EVALUATE
13	0 0.0%	9	Scan	0	0		3	1		SELECT 'Sa

The DirectQuery query executed is way simpler than before:

```

--
-- DAX DirectQuery query executed on the remote model
--
DEFINE
    VAR _Var0 = VALUES ( 'Date'[Year] )
    VAR _Var1 = VALUES ( 'Customer'[CustomerKey] )
    VAR _Var2 = VALUES ( 'Customer'[CustomerKey] )
EVALUATE
    GROUPOSSAPPLYTABLE (
        'Date'[Year],
        _Var0,
        _Var1,
        "L1",
        GROUPOSSAPPLY (
            'Customer'[CustomerKey],
            KEEPFILTERS ( _Var2 ),
            "__Agg0", [Sales Amount]
        )
    )

```

This query retrieves the value of *Sales Amount* for each customer and year, exactly as the previous query did. The big difference is that we no longer have the ridiculous filter – instead, we obtain a smaller query, and the result set contains only the customer key, the year, and the sales amount. Consequently, the resulting data cache is smaller.

As you have seen, we updated the DAX code specifically for a composite model in this example. A measure that works just fine on a regular VertiPaq model might show performance issues with a composite model. A similar scenario applies with DirectQuery models: VertiPaq and a relational database like SQL Server are very different engines, so the DAX code optimization depends on the storage engine(s) used.

Conclusions

The goal of this chapter was not to teach any technique. We optimized DAX code, processing, and models without describing the rationale behind each decision in detail. The important part was to show that optimizing a Tabular model requires knowing several details and using many different tools. Without proper knowledge about the internals, it is nearly impossible to produce efficient code. A good DAX developer should master each of these techniques and tools.

Starting from the next chapters, we go into the details and build the knowledge needed to optimize your Tabular models. It will take some time before we can put the learning into practice. Even though at some point you might feel that all the details we provide are not so relevant... Take a deep breath and continue ingesting knowledge. As soon as we start to optimize, you will see that each detail is helpful.

Get the full book on

[**https://sql.bi/optdax**](https://sql.bi/optdax)