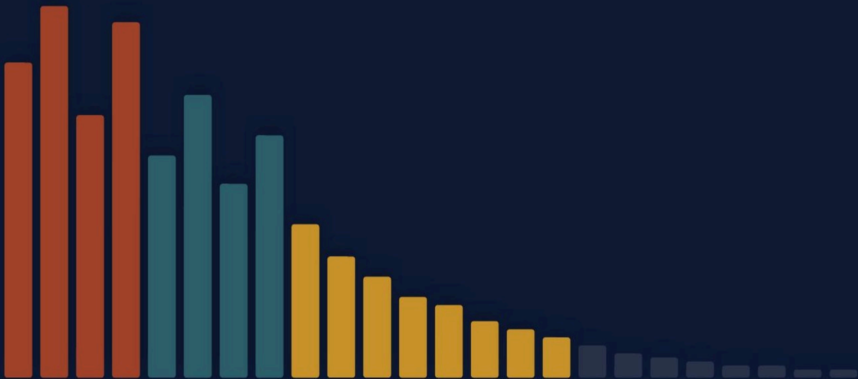
FIRST EDITION · 2025

THE Power BI *Performance* Bible

Model · DAX · Visuals · Scale

MAKE YOUR REPORTS FAST ENOUGH TO ACTUALLY USE



Syed Hussnain Tahir Sherazi

DATA ENGINEER

VOL. I

THE PRACTITIONER'S FIELD MANUAL

EIGHT CHAPTERS · THREE APPENDICES

VOLUME ONE · THE PRACTITIONER'S FIELD MANUAL

THE

Power BI *Performance* Bible

Model · DAX · Visuals · Scale

MAKE YOUR REPORTS FAST ENOUGH TO ACTUALLY USE

VOLUME ONE · FIRST EDITION · 2025

THE

Power BI *Performance* Bible

Model · DAX · Visuals · Scale

A PRACTITIONER'S FIELD MANUAL

Syed Hussnain Tahir Sherazi

DATA ENGINEER · LEICESTER · 2025

contact@syedhussnain.com · linkedin.com/in/hussnainsherazi

Copyright & Disclaimer

The Power BI Performance Bible: Model, DAX, Visuals and Scale

Copyright © 2025 Syed Hussnain Tahir Sherazi. All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the author, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law.

Microsoft, Power BI, DAX, Power Query, and related trademarks are the property of Microsoft Corporation. This book is an independent publication and is not affiliated with, nor has it been authorised, sponsored, or otherwise approved by Microsoft Corporation.

The information in this book is provided on an "as is" basis. The author makes no warranties regarding the accuracy or completeness of the contents. Advice and strategies contained herein may not be suitable for every situation. Readers should test all changes in a non-production environment first.

PUBLISHER · EDITORIAL

Sherazi Press · 2025 — Leicester · United Kingdom

First Edition

Contents

VOL. I · 8 CHAPTERS

–	*Copyright & Disclaimer*	04
–	*About the Author*	06
–	*How to Use This Book*	07
–	*Tools Referenced*	08
–	*Preface · The Report That Made Everyone Wait*	09
CH.01	How Power BI Actually Works	11
CH.02	Data Model — The Foundation	16
CH.03	DAX Performance	25
CH.04	Power Query & Refresh Performance	36
CH.05	Visuals & Report Design	42
CH.06	DirectQuery & Composite Models	47
CH.07	Deployment, Capacity & Service	53
CH.08	The Performance Optimisation Playbook	61
–	*Appendix A · DAX Quick Reference*	68
–	*Appendix B · Power Query Folding Cheat Sheet*	69
–	*Appendix C · Glossary*	70
–	*A Final Word from the Author*	72

THE AUTHOR

Syed Hussnain Tahir Sherazi

DATA ENGINEER · LEICESTER · 2025

Syed Hussnain Tahir Sherazi is a Microsoft Certified Business Intelligence Development expert with deep proficiency in Power BI and Microsoft Fabric. He has built his career in environments where mistakes in data are not really a choice. He earned his early stripes in the global smartphone industry, working across brands including Realme, Infinix, Tecno, and Itel, where data driven decision making shaped product launches, user experience research, and after sales performance across Pakistan, Africa, and China. Later he moved to the UK and stepped into the public sector, where data is rough, inherited, and unforgiving, and where reports face directors, elected members, and the general public. He currently works at Leicester City Council, building end to end Power BI solutions, contributing to the migration from on premises SQL Server to Microsoft Fabric, and authoring the council's official Power BI Style Guide. His work focuses on the unglamorous side of business intelligence: model design, DAX discipline, query folding, and the daily habits that decide whether a report is trusted or quietly abandoned. He holds a Master's in Cloud Computing from the University of Lincoln and a Bachelor's in Software Engineering from the University of Management and Technology. He writes from Leicester.

✉ contact@syedhussnain.com
● linkedin.com/in/hussnainsherazi

FRONT MATTER

How to use this book.

This book is written for people who build Power BI reports and models every day. Business analysts, BI developers, data leads — if you have ever stared at a spinning wheel and wondered why your report takes so long, this book is for you.

You do not need to read it cover to cover. Each chapter stands on its own. If your model is slow, start with Chapter 2. If your DAX is the problem, jump to Chapter 3. If you just need a quick checklist before publishing, go straight to Chapter 8.

Every chapter ends with a short *Try this* section. These are hands-on exercises you can do with your own reports. They are not homework — they are the fastest way to see results.

DAX code examples use a consistent naming convention throughout. Tables like `Sales`, `Customer`, `Product`, and `'Date'` appear in most examples. When you see a **SLOW** label, that is code that works but performs badly. The **FAST** label shows the better alternative. Both are real, runnable DAX.

> *Most slow Power BI reports can be made fast in less than a day. You just need to know where to look — and what to change.*

FRONT MATTER

Tools you'll need.

This book references several free and essential tools for Power BI performance work. None of them cost anything. All of them belong in your tool belt.

DAX Studio

A free tool for running DAX queries, capturing server timings, and analysing the VertiPaq storage engine. Download from `daxstudio.org`. You will use this tool extensively in Chapters 3 and 8.

Tabular Editor

An open-source editor for tabular models. Version 2 is free. It lets you inspect and edit your model outside Power BI Desktop, run Best Practice Analyser rules, and script changes. Essential for Chapter 2 model audits.

Performance Analyser

Built into Power BI Desktop under the *View* tab. It shows you exactly how long each visual takes to render, how long the DAX query takes, and how long the visual itself takes to draw. Chapter 5 walks through it step by step.

Fabric Capacity Metrics App

A Microsoft-provided app for monitoring Premium and Fabric capacity usage. It shows throttling, queuing, and which datasets consume the most resources. Chapter 7 covers installation and key metrics.

FRONT MATTER

The Report That Made Everyone Wait.

It was a Tuesday morning. The sales director opened the weekly pipeline report. She clicked one slicer. The screen went blank. The spinning circle appeared. She waited. Ten seconds. Twenty seconds. Thirty seconds. At forty-five seconds, she closed the browser tab and opened a spreadsheet instead.

That spreadsheet was three weeks out of date. She made decisions based on old numbers. Nobody told the BI team. The report kept running in the background, consuming capacity, delivering value to no one.

I have seen this story play out dozens of times. The details change — different company, different report, different slicer — but the outcome is always the same. Slow reports kill adoption. And dead adoption is the most expensive outcome in business intelligence.

Think about what it costs to build a Power BI solution. Weeks of requirements gathering. Data modelling. DAX measures. Testing. Training. Deployment. All of that investment evaporates the moment a user decides the report is too slow to use.

> *Performance is not a technical nice-to-have. Performance is a trust issue.*

When a report loads in two seconds, users trust the data. They explore. They drill down. They come back tomorrow. When a report takes forty-five seconds, users do not trust anything about it. They assume the data is wrong too. Why wouldn't they? If we cannot get the loading time right, why would anyone believe we got the numbers right?

This book is about fixing that. Not in theory. In practice. With real DAX code, real model changes, and real diagnostic steps you can follow today.

I wrote this book because I got tired of explaining the same fixes in every engagement. The same `SUMX` that should be a `SUM`. The same bidirectional relationship that nobody needs. The same twenty slicers on one page. The same calculated columns that should be measures. The patterns repeat because the fundamentals are not widely understood.

The good news is that most performance problems are fixable. I would estimate that eighty percent of slow Power BI reports can be made fast with changes that take less than a day. You do not need to rebuild anything. You need to know where to look and what to change.

That is what this book will teach you. Let's make your reports fast enough that people actually use them.

— *Syed Hussnain Tahir Sherazi*

Leicester, May 2025

CHAPTER ONE

01

How Power BI Actually Works.

Before you can fix a slow report, you need to understand what happens between the click and the chart. Most performance problems become obvious once you understand the machinery underneath.

THREE ENGINES | QUERY FLOW | SE VS FE | COMPRESSION

The three storage modes

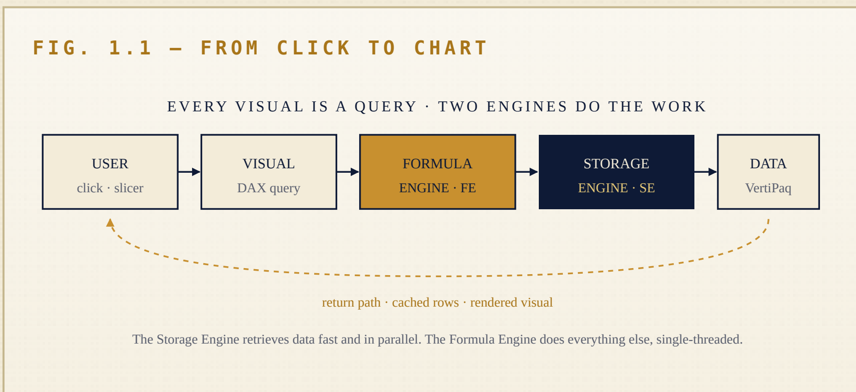
Power BI offers three storage modes, and each one changes how queries are processed.

- **VertiPaq (Import Mode)** loads data into memory, compresses it, and stores it inside the `.pbix` file. The query runs against this in-memory copy. No connection to the original source is needed at query time. Fastest for most scenarios.
- **DirectQuery** stores nothing locally. Every interaction sends a query back to the source database. Real-time, but every click depends on the source's speed.
- **Composite Models** combine both. Import some tables to VertiPaq, keep others in DirectQuery. Powerful, but the engine must decide at runtime which mode handles each part of the query.

For the rest of this book, unless stated otherwise, we are talking about Import Mode. It is the most common and the mode where you have the most control over performance.

How a query travels

FIG. 1.1 — FROM CLICK TO CHART

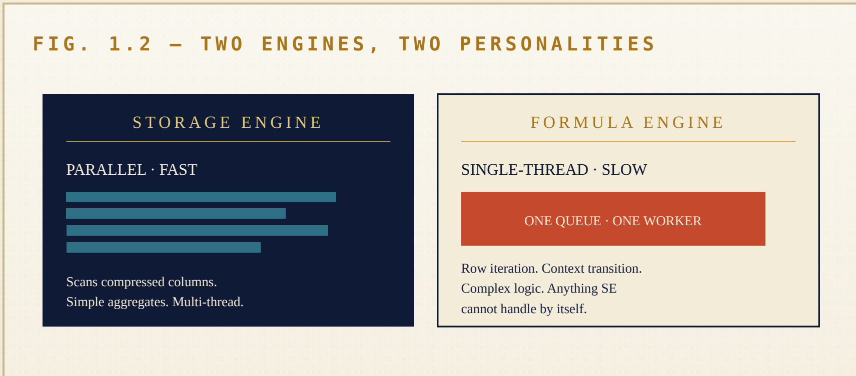


The Storage Engine retrieves data fast and in parallel. The Formula Engine does everything else, single-threaded.

Every time a user interacts with a Power BI report, a DAX query is generated. The important thing to notice: there are **two engines** involved, not one.

Storage Engine vs Formula Engine

FIG. 1.2 — TWO ENGINES, TWO PERSONALITIES



The **Storage Engine (SE)** is fast because it runs in parallel across multiple threads. It works with compressed, columnar data and is optimised for scanning and aggregating large datasets. When the SE can handle a query entirely by itself, the result comes back in milliseconds.

The **Formula Engine (FE)** is single-threaded. It handles everything the SE cannot: complex calculations, row-by-row iteration, context transitions. When too much work lands on the FE, performance suffers because it processes one operation at a time.

> *Push as much work as possible to the Storage Engine. Keep the Formula Engine light.*

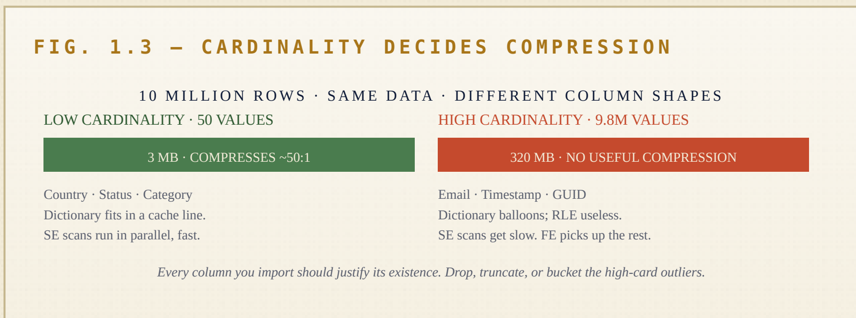
In practice, this means writing DAX that can be translated into simple storage-engine scans rather than row-by-row iterations. Chapter 3 covers the specific patterns.

The role of compression

VertiPaq compresses data using two main techniques:

- **Dictionary encoding.** Each unique value in a column is stored once in a dictionary. Every row stores a short reference number instead of the full value. If a column has 10 million rows but only 500 unique values, the dictionary has 500 entries and each row stores a tiny integer.
- **Run-Length Encoding (RLE).** When consecutive rows share the same value, VertiPaq stores the value once plus a count. A column with 1 million rows of "Active" followed by 200,000 rows of "Inactive" compresses down to just two entries.

This is why **cardinality matters so much**. A column with 50 unique values compresses beautifully. A column with 5 million unique values — a full timestamp, a free-text email field — barely compresses at all. It takes up more memory, slows down scans, and increases query time.



The practical lesson: every column you import should justify its existence. If a column has high cardinality and nobody uses it in a report, remove it. If a `DateTime` column is only used for date-level analysis, truncate it to `Date`. These are not minor tweaks. They can reduce model size by 30–60%.

Quick diagnostic checklist

When a report is slow, use this table to narrow down where the problem is before diving into details:

SYMPTOM	LIKELY CAUSE	READ
Entire report loads slowly	Too many visuals or model too large	Ch 2 · Ch 5
One specific visual is slow	Complex DAX measure or high-cardinality column	Ch 3
Slicer selection causes long wait	High-card slicer or cross-filter chain	Ch 5
Data refresh takes hours	Query folding broken or too much data imported	Ch 4
Fast in Desktop, slow in Service	Capacity throttling or RLS overhead	Ch 7
DirectQuery report always slow	Source database unoptimised, no aggregations	Ch 6
Model file size enormous	High-cardinality or unnecessary columns	Ch 2

TRY THIS

1. Open a Power BI report you work with regularly. Go to *View > Performance Analyser* and click **Start Recording**.
2. Interact with the report: change a slicer, click a chart, switch pages. Stop the recording and look at the results.
3. For each visual, note DAX query time and visual rendering time. Write down the three slowest visuals and their query times. You will refer to this list throughout the book.

CHAPTER TWO

02

Data Model — The Foundation.

Your model is the foundation of everything. A bad model makes good DAX slow. A good model makes simple DAX fast. If you only optimise one thing — optimise this.

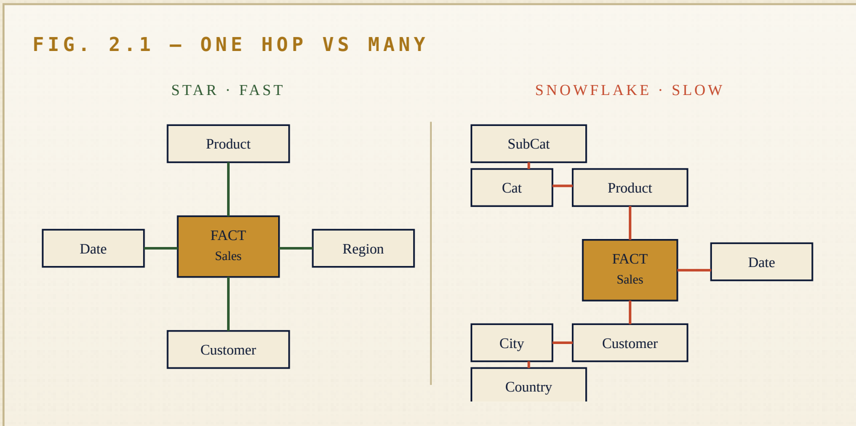
STAR SCHEMA | CARDINALITY | RELATIONSHIPS | DATE TABLES

Star schema vs everything else

A star schema has one central fact table surrounded by dimension tables. Each dimension connects to the fact with a single one-to-many relationship. Fact tables store measurements (sales amounts, quantities). Dimensions store descriptive attributes (product names, customer details, dates).

A snowflake adds extra tables between dimensions. Instead of a single Product dimension, you might have Product → SubCategory → Category. Each extra table adds a relationship the engine must traverse at query time.

FIG. 2.1 — ONE HOP VS MANY



Every extra relationship in the chain adds query cost. The engine must join across more tables, reducing its ability to compress and scan efficiently. Flatten your snowflake dimensions into star schema dimensions. Add the `SubCategory` and `Category` columns directly to the Product table.

The cardinality problem

Cardinality is the number of unique values in a column. Low-cardinality columns (Status with 5 values, Country with 50) compress well. High-cardinality columns (timestamps with millions of unique values, email addresses, GUIDs) compress poorly and inflate model size.

COMMON OFFENDERS & THEIR FIXES

COLUMN TYPE	PROBLEM	FIX
DateTime (timestamps)	Every row unique to the second or millisecond	Truncate to `Date`. Split time to a separate column if needed.
Email addresses	Nearly one unique value per row	Remove unless used in filters. If needed, keep in a small lookup table.
Free-text fields	Every entry unique	Remove from model. Store in a drill-through detail table.
GUIDs / surrogate keys	Unique by design	Use integer keys. If only used for relationships, ensure they are integers.
Decimal with high precision	Many unique values from precision	Round to 2 decimal places unless precision is required.

Column discipline

Only import columns you actually use — in visuals, measures, relationships, or row-level security. Every extra column consumes memory, slows refresh, and makes scans take longer.

Use DAX Studio's **VertiPaq Analyser** to audit your model. Connect to your model and run the analyser from the *Advanced* menu. It shows every table and column with its size, cardinality, and encoding type.

Key things to investigate:

- Columns with cardinality close to row count — almost always removable.
- Columns taking up more than 10% of a table's total size.
- Columns not referenced in any measure or visual.

FIND UNUSED COLUMNS · DAX

```
EVALUATE
SELECTCOLUMNS (
    INFO.COLUMNS (),
    "Table",  [TableName],
    "Column", [ColumnName],
    "Size",   [ColumnSize]
)
ORDER BY [Size] DESC
```

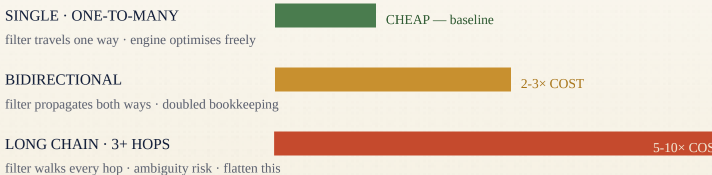
Cross-reference this list with your report's actual field usage. Any column that appears in the model but not in any visual, measure, or relationship is a candidate for removal.

Relationships done right

Every relationship has a cost. Single-direction, one-to-many relationships are cheapest. Bidirectional relationships are more expensive because the engine must maintain filter propagation in both directions.

- **Bidirectional filtering** should be used only when absolutely necessary — many-to-many bridge table patterns, mostly. Every bidirectional relationship doubles the filter work and can cause ambiguous filter paths.
- **USERELATIONSHIP** in DAX activates an inactive relationship at query time. Useful for role-playing dimensions (a Date table linked to both `OrderDate` and `ShipDate`), but each call adds overhead. Keep role-playing relationships to two or three at most.
- **Long chains** — where a filter must travel through three or more tables to reach the fact — are slow. If you find yourself chaining more than two relationships, flatten the model.

FIG. 2.2 — RELATIONSHIP COSTS



Date tables

Every Power BI model needs a proper date table. Without one, time-intelligence functions either fail or produce incorrect results, and performance suffers because the engine cannot optimise date-based queries.

A proper date table must be **continuous** (every day from earliest to latest, no gaps), **marked** as a date table (right-click → *Mark as Date Table*), and linked to your fact table's date column via a single one-to-many relationship.

Do not add a time component to the `Date` column. Keep it as `Date` only, not `DateTime`. This keeps cardinality low and compression high.

> *If your model has a Date column with millisecond precision, you have two problems: a slow report, and a date table you don't actually trust.*

The next page shows the recommended columns to include. Build this table once, save it in your standard model template, and reuse it across projects.

Recommended date table columns

COLUMN	TYPE	EXAMPLE
Date	Date	2025-01-15
Year	Whole Number	2025
Quarter	Text	Q1
MonthNumber	Whole Number	1
MonthName	Text	January
WeekNumber	Whole Number	3
DayOfWeek	Whole Number	4 (Wednesday)
DayOfWeekName	Text	Wednesday
IsWeekend	Boolean	FALSE
FiscalYear · FiscalQuarter	Whole / Text	2025 · FQ3

Row count vs column count trade-off

In a columnar database like VertiPaq, adding rows is relatively cheap if those rows reuse values already in the column dictionaries. Adding columns is expensive because each column needs its own dictionary, encoded data, and memory allocation.

A table with 10 million rows and 8 columns will often perform better than a table with 2 million rows and 40 columns — especially if many of those 40 columns are high-card. When in doubt, go narrow. Remove columns you do not need. Split wide fact tables into a core fact and detail tables linked by key.

> *Rows are cheap. Columns are expensive. When in doubt, go narrow.*

Model sizing & licence limits

Power BI imposes size limits depending on your licence:

LICENCE	MAX DATASET	LARGE FORMAT	REFRESH / DAY
Power BI Pro	1 GB	Not available	8
Premium Per User	10 GB	10 GB	48
Premium / Fabric	10 GB default	Up to capacity memory	Unlimited (XMLA)

If your model is approaching the limit for your licence tier, **do not** upgrade your licence as the first response. Audit the model first. In most cases, removing unused columns, reducing cardinality, and disabling auto date/time tables will bring the size down significantly.

> *The cheapest gigabyte of capacity is the one you don't need to buy.*

Practical model audit · 7 steps

1. **Open DAX Studio** and connect to your model. *Advanced > View Metrics* to open VertiPaq Analyser.
2. **Tables tab** — sort by Total Size descending. Identify largest tables. Biggest payoff lives here.
3. **Columns tab** — sort by Total Size descending. Look for columns taking disproportionate space. Check cardinality. Cardinality close to row count = removal candidate.
4. **Check auto date/time tables.** File > Options > Current File > Data Load → uncheck Auto Date/Time. Each hidden date table adds roughly 6,000 rows.
5. **Review relationships.** Model view. Look for bidirectional (double arrows). For each: does it really need to be bidirectional?
6. **Run Best Practice Analyser** in Tabular Editor. Tools > BPA. Flags unused columns, bidirectional relationships, missing sort-by columns.
7. **Document findings.** Current model size, top five tables, top ten columns. After changes, re-run and compare.

TRY THIS

1. Pick a Power BI model you work with. Open in DAX Studio and run VertiPaq Analyser.
2. Write down: total model size, largest table, highest-cardinality column, count of auto date/time tables.
3. Remove one unused high-cardinality column. Save, refresh, re-run the analyser. Compare. Celebrate.

CHAPTER THREE

03

DAX *Performance.*

A single badly written measure can turn a two-second report into a forty-five-second report. Ten patterns. Ten fixes. One discipline.

CONTEXT | CALCULATE | ANTI-PATTERNS | VAR / RETURN

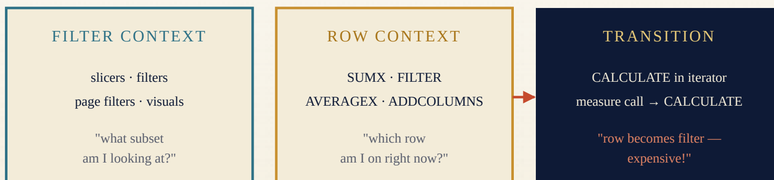
How DAX is evaluated

Every DAX expression is evaluated within a context. There are two types: filter context and row context.

- **Filter context** is the set of active filters applied to the data. When a user selects "2025" in a year slicer, that creates a filter context that limits all calculations to rows where year is 2025. Every visual operates within filter context.
- **Row context** exists inside iterator functions like `SUMX`, `AVERAGEX`, and `FILTER`. The engine creates a row context that evaluates the expression for each row.
- **Context transition** occurs when `CALCULATE` (or a measure reference) is used inside a row context. The row context is converted into an equivalent filter context. Powerful but expensive — every transition triggers a new query against the model.

`CALCULATE` is the most important function in DAX for performance. It does three things: it changes filter context, it triggers context transition, and it forces the engine to re-evaluate an expression under new conditions. Every unnecessary `CALCULATE` adds overhead.

FIG. 3.1 — THE THREE CONTEXTS



Every transition fires a new query. Use them when you must — never by accident.

The top 10 DAX anti-patterns

For each anti-pattern below, the slow version is shown first, followed by the fast alternative.

1 · SUMX OVER LARGE TABLES

Iterating row by row when a simple `SUM` would suffice.

SLOW

```
[Total Revenue] =
SUMX (
    Sales,
    Sales[Amount]
)
-- iterates every row · unnecessary when summing a column
```

FAST

```
[Total Revenue] =
SUM ( Sales[Amount] )
-- handled entirely by the Storage Engine · no iteration
```

2 · NESTED CALCULATE

Wrapping `CALCULATE` inside `CALCULATE` without good reason.

SLOW

```
[Sales Last Year] =
CALCULATE ( CALCULATE (
    SUM ( Sales[Amount] ),
    SAMEPERIODLASTYEAR ( 'Date'[Date] )
) )
```

FAST

```
[Sales Last Year] =
CALCULATE (
    SUM ( Sales[Amount] ),
    SAMEPERIODLASTYEAR ( 'Date'[Date] )
)
```

3 · IF WITH MEASURE REFERENCES

SLOW

```
[Display Sales] =
IF (
    ISBLANK ( [Total Sales] ),
    0,
    [Total Sales]
)
-- IF evaluates BOTH branches
```

FAST

```
[Display Sales] =
COALESCE ( [Total Sales], 0 )
-- returns first non-blank · single evaluation
```

4 · FILTER ON WHOLE TABLE VS COLUMN FILTER

SLOW

```
[Online Sales] =
CALCULATE (
    SUM ( Sales[Amount] ),
    FILTER ( Sales, Sales[Channel] = "Online" )
)
-- iterates entire Sales table
```

FAST

```
[Online Sales] =
CALCULATE (
    SUM ( Sales[Amount] ),
    Sales[Channel] = "Online"
)
-- column filter · Storage Engine handles it
```

5 · DISTINCTCOUNT ON HIGH CARDINALITY

SLOW

```
[Unique Transactions] =
DISTINCTCOUNT ( Sales[TransactionGUID] )
-- GUIDs are millions of unique strings
```

FAST

```
[Unique Transactions] =
DISTINCTCOUNT ( Sales[TransactionID] )
-- integer key · lower cardinality · better compression
```

6 · OVERUSING ALLEXCEPT

SLOW

```
[Category Share] =
DIVIDE (
    SUM ( Sales[Amount] ),
    CALCULATE (
        SUM ( Sales[Amount] ),
        ALLEXCEPT ( Sales, Sales[ProductCategory] )
    )
)
-- removes filters from every unlisted column
```

FAST

```
[Category Share] =
VAR _total =
    CALCULATE (
        SUM ( Sales[Amount] ),
        REMOVEFILTERS ( 'Product'[ProductName] ),
        REMOVEFILTERS ( 'Product'[SubCategory] )
    )
RETURN
    DIVIDE ( SUM ( Sales[Amount] ), _total )
```

7 · TIME INTELLIGENCE ON NON-MARKED DATE TABLES

The DAX looks identical. The engine performance does not. A marked date table unlocks an optimised internal path for every time-intelligence function.

SLOW · UNMARKED

```
[YTD Sales] =
TOTALYTD ( SUM ( Sales[Amount] ), 'Date'[Date] )
-- engine cannot use the optimised path · slow
```

FAST · MARKED AS DATE TABLE

```
[YTD Sales] =
TOTALYTD ( SUM ( Sales[Amount] ), 'Date'[Date] )
-- right-click Date table → Mark as Date Table
-- same DAX · 5-10× faster execution
```

A simple discipline

If you only fix one thing in your model today, mark your date table. The DAX you wrote yesterday will get faster overnight, with no rewrites. There is no other free performance win quite like it.

Once you have marked the date table, every `TOTALYTD`, `SAMEPERIODLASTYEAR`, `DATEADD`, and friends route through the engine's optimised time-intelligence path. The difference shows up immediately in Server Timings — fewer SE queries, lower FE time, often a halved total.

> *You will rarely find a model in the wild where the date table has been properly marked. Mark yours. Then go check every model your team owns.*

8 · CHAINED MEASURES WITHOUT VAR CACHING

SLOW

```
[Sales Status] =
IF (
    [Total Sales] > 1000000,
    "High: " & FORMAT ( [Total Sales], "#,0" ),
    "Low: "  & FORMAT ( [Total Sales], "#,0" )
)
-- [Total Sales] evaluated THREE times
```

FAST

```
[Sales Status] =
VAR _sales = [Total Sales]
RETURN
    IF (
        _sales > 1000000,
        "High: " & FORMAT ( _sales, "#,0" ),
        "Low: "  & FORMAT ( _sales, "#,0" )
    )
-- evaluated once · reused
```

9 · RANKX OVER LARGE TABLES

SLOW

```
[Product Rank] =
RANKX ( ALL ( 'Product' ),
    CALCULATE ( SUM ( Sales[Amount] ) ) )
-- re-evaluates for every row
```

FAST

```
[Product Rank] =
VAR _table =
    ADDCOLUMNS (
        ALL ( 'Product'[ProductName] ),
        "@Sales", CALCULATE ( SUM ( Sales[Amount] ) )
    )
VAR _current = [Total Sales]
RETURN
    COUNTROWS ( FILTER ( _table, [@Sales] > _current ) ) + 1
```

10 · CALCULATED COLUMNS INSTEAD OF MEASURES

This is the most common offender of all. Calculated columns are stored row-by-row in the model, inflating size and refresh time. Measures are computed at query time and cost nothing to store.

SLOW

```
Sales[Revenue] = Sales[Quantity] * Sales[UnitPrice]
-- stored in memory for every row
-- inflates table size · slows refresh
-- recalculated on every model change
```

FAST

```
[Revenue] = SUMX ( Sales, Sales[Quantity] * Sales[UnitPrice] )
-- computed at query time
-- zero storage cost
-- only runs when a visual needs it
```

Use calculated columns only for three things: sorting (a derived sort-by column), relationships (a key built from two columns), or row-level security (a flag the engine needs at row level). For everything else — use measures.

> *If you can write it as a measure, you should. The model gets smaller, the refresh gets faster, and your reports stay flexible.*

Variables

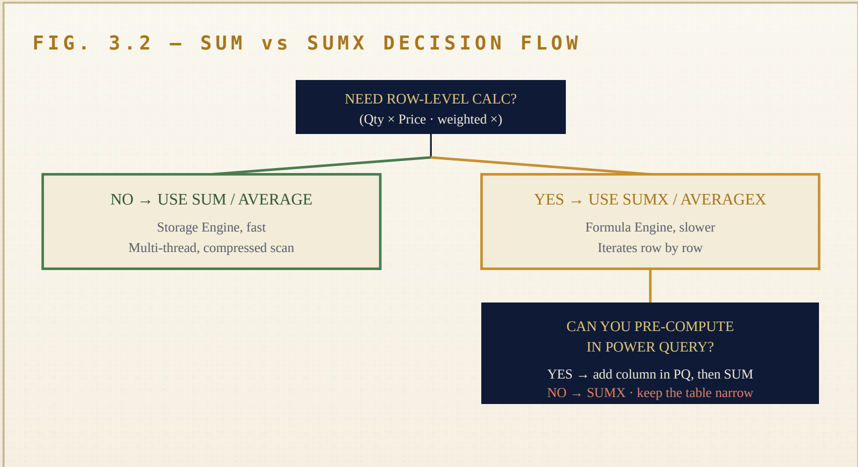
The `VAR` / `RETURN` pattern is the single most useful DAX habit for both performance and readability. Variables are evaluated once at the point of definition and then reused.

CANONICAL PATTERN

```
[MoM Growth %] =
VAR _currentMonth  = [Total Sales]
VAR _previousMonth =
    CALCULATE ( [Total Sales], DATEADD ( 'Date'[Date], -1, MONTH ) )
VAR _growth = _currentMonth - _previousMonth
RETURN
    DIVIDE ( _growth, _previousMonth )
```

Here `[Total Sales]` is evaluated exactly twice — once per period. Without variables, you might accidentally evaluate it four or five times.

Aggregation vs iteration · the decision



Context transition cost

Context transition occurs when a measure is evaluated inside an iterator. The row context is silently converted to filter context via an implicit `CALCULATE`.

DANGEROUS · TRIGGERS TRANSITION PER ROW

```
[Weighted Sales] =
SUMX ( Sales, [Total Sales] * Sales[Weight] )
```

FIX · USE COLUMN REFERENCES, NOT MEASURES

```
[Weighted Sales] =
SUMX ( Sales, Sales[Amount] * Sales[Weight] )
```

RELATED vs LOOKUPVALUE. `RELATED` follows an existing relationship and is fast. `LOOKUPVALUE` performs a lookup without one and is slower. Prefer `RELATED`.

Measuring DAX with DAX Studio

1. Open DAX Studio and connect to your Power BI Desktop model.
2. Go to *Traces* and enable **Server Timings**.
3. Copy the DAX query from Performance Analyser (right-click slow visual → *Copy query*).
4. Paste and run. Look at the Server Timings pane.

METRIC	WHAT IT MEANS
Total	Total query execution time in milliseconds.
SE · Storage Engine	Time scanning data. Parallel — usually fast.
FE · Formula Engine	What SE could not handle. Single-threaded — usually the bottleneck.
SE Queries	Storage-engine query count. More than 10–20 suggests too many scans.

If results show `Total 4,200ms · SE 800ms · FE 3,400ms`, the bottleneck is in the Formula Engine. Simplify, add variables, reduce transitions.

Optimising time intelligence

Time-intelligence functions vary in cost:

FUNCTION	COST	NOTES
`TOTALYTD / TOTALQTD / TOTALMTD`	Low	Simple date-range filter
`SAMEPERIODLASTYEAR`	Low	Shifts range by one year
`DATEADD`	Low-Medium	Flexible shift, slight overhead
`PARALLELPERIOD`	Medium	Watch for partial periods
`DATESBETWEEN`	Medium	Explicit range, higher cost

All time-intelligence functions are faster when the date table is properly marked. For complex calculations, build them step by step with variables:

ROLLING 12 MONTHS · STEP BY STEP

```
[Rolling 12 Month Sales] =
VAR _lastDate  = MAX ( 'Date'[Date] )
VAR _startDate = DATE ( YEAR (_lastDate) - 1,
                        MONTH (_lastDate) + 1, 1 )
RETURN
    CALCULATE (
        SUM ( Sales[Amount] ),
        DATESBETWEEN ( 'Date'[Date], _startDate, _lastDate )
    )
```

TRY THIS

1. Pick the slowest measure in your report. Copy its query from Performance Analyser into DAX Studio.
2. Enable Server Timings, run it, write down SE time, FE time, SE query count.
3. Does the measure use any of the ten anti-patterns? Rewrite using the fast alternative. Re-run and compare. In most reports, fixing the top three slow measures cuts overall load time by 40–60%.

CHAPTER FOUR

04

Power Query & Refresh.

If your gateway queues back up and your users see stale data, Folding is the answer. The right Power Query order saves hours per refresh.

FOLDING | DATA TYPES | MERGE | INCREMENTAL REFRESH

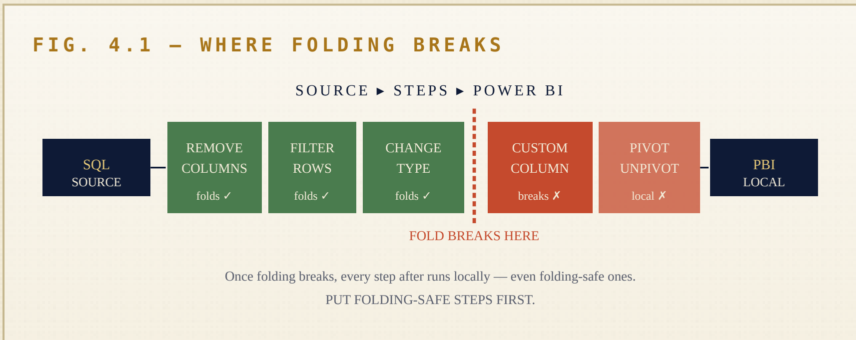
Query folding

Query folding is when Power Query translates your M code steps into native queries that run on the data source. Instead of downloading all the data and filtering locally, the source database does the filtering and only sends back the rows you need.

This is the single most important concept in Power Query performance. **A folded query can be 10 to 100 times faster than an unfolded one.**

To check whether a step folds, right-click on it in the *Applied Steps* pane. If you see "View Native Query," the step folds. If the option is greyed out, folding has broken at that point, and every step after it runs locally.

FIG. 4.1 — WHERE FOLDING BREAKS



Once folding breaks, every step after runs locally — even folding-safe ones.
PUT FOLDING-SAFE STEPS FIRST.

Folding-safe vs folding-breaking steps

FOLDING-SAFE	FOLDING-BREAKING
Remove Columns	Add Column from Examples
Filter Rows	Merge queries (cross-source)
Sort Rows	Pivot / Unpivot (most sources)
Rename Columns	Group By (some sources)
Change Data Type	Add Custom Column with M logic
Replace Values (simple)	`Table.Buffer`
Keep / Remove Top / Bottom Rows	`Text.Combine` with custom logic
Choose Columns	Running totals · index columns

> *The golden rule: put folding-safe steps first. Once a folding-breaking step appears, everything after it runs locally.*

Right data types from the start

Data types matter more than most people realise. A `DateTime` column has far higher cardinality than a `Date` column because every unique time value creates a new dictionary entry in VertiPaq.

Set data types as early as possible in your Power Query steps. If you need only the date, change `DateTime` to `Date` in Power Query — not after import. This ensures the data enters the model with low cardinality from the start.

Common type decisions

COLUMN	WRONG	RIGHT	WHY
`OrderDate`	DateTime	Date	Reduces cardinality by 86,400×
`Quantity`	Decimal	Whole Number	Smaller storage, better compression
`PostalCode`	Whole Number	Text	Leading zeros · not really numbers
`Amount`	Text	Fixed Decimal	Enables aggregation, reduces size

Remove before you transform

The order of steps in Power Query matters for both folding and memory. Follow this sequence:

1. Remove columns you do not need. Reduces width.
2. Filter rows to keep only what you need. Reduces volume.
3. Set data types.
4. Apply transformations and custom columns.

WELL-ORDERED M QUERY

```
let
    Source = Sql.Database("server", "database"),
    Sales  = Source{[Schema="dbo", Item="Sales"]}[Data],

    // 1 · Remove columns first (folds)
    Removed = Table.SelectColumns(Sales,
        {"OrderDate","CustomerKey","Quantity","Amount"}),

    // 2 · Filter rows (folds)
    Filtered = Table.SelectRows(Removed,
        each [OrderDate] >= #date(2024, 1, 1)),

    // 3 · Set data types (folds)
    Typed = Table.TransformColumnTypes(Filtered,
        {{"OrderDate", type date}, {"Amount", Currency.Type}})
in
    Typed
```

Calculated column or Power Query column?

You can compute a derived value in two places: a calculated column in DAX (after import) or a custom column in Power Query (during refresh). Both are stored permanently — memory cost is similar. The key difference is timing.

USE POWER QUERY WHEN…	USE DAX WHEN…
Calculation uses only the current table	Calculation needs `RELATED` data
Logic is simple (concat, type, bucket)	Logic needs DAX functions not in M
You want it to fold to the source	Column is used for context-aware RLS

Merging queries efficiently

Merge in Power Query is equivalent to a SQL JOIN. The key question is whether both sides come from the same source.

- **Same-source merges** can fold to a server-side JOIN. Fast — the database does the work.
- **Cross-source merges** cannot fold. Power Query downloads both tables into memory and joins locally. On large tables, slow and memory-heavy.

If you must merge across sources, minimise the data first. Filter rows and remove columns from both tables before the merge. If one table is small (a few hundred rows), the cross-source merge is tolerable. If both have millions of rows, move the join to the source database using a view.

Join types: inner and left outer are most efficient. Full outer and cross join produce larger intermediate results — avoid unless truly needed.

Incremental refresh

Incremental refresh tells Power BI to only refresh the data that has changed, rather than reloading the entire table every time. On large datasets, this can reduce refresh time from hours to minutes.

Setup requires two parameters in your query: `RangeStart` and `RangeEnd`, both of type `DateTime`. Power Query uses these to filter your source data. Power BI then manages partitions behind the scenes, refreshing only the most recent partition.

INCREMENTAL REFRESH · REQUIRED FILTER PATTERN

```
let
    Source = Sql.Database("server", "database"),
    Sales  = Source{[Schema="dbo", Item="Sales"]}[Data],

    // This filter pattern is required — do not modify
    Filtered = Table.SelectRows(Sales, each
        [OrderDate] >= RangeStart and [OrderDate] < RangeEnd)
in
    Filtered
```

When to use it: table has more than 1 million rows, source supports folding on the date filter, you have a reliable date column.

Partition size: daily for tables growing by 100k+ rows per day. Monthly for smaller tables. Keep the refresh window (recent partitions refreshed each time) as small as reasonable — typically 1 to 7 days.

TRY THIS

1. Open Power Query in your model. For each table, right-click the last step in *Applied Steps*. Check if "View Native Query" is available.
2. Work backwards until you find the last step that folds. The step after is where folding breaks. Can you reorder?
3. Check column counts — for each table, how many columns are imported versus available in the source? If you import more than half, you probably import too many.

CHAPTER FIVE

05

Visuals & Report Design.

Every visual is a filter. Every visual is a query. Every interaction fires new calculations. Report design is performance design.

VISUAL COUNT | SLICERS | CROSS-FILTER | PERFORMANCE ANALYSER

How visuals generate queries

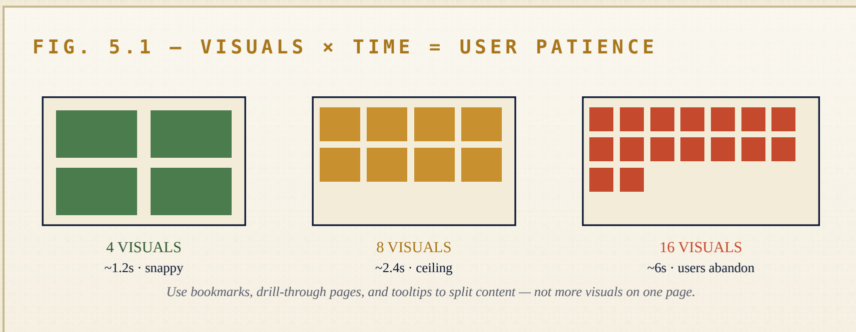
When a report page loads, Power BI generates a separate DAX query for every visual on the page. A page with twelve visuals fires twelve queries. If each takes 300 milliseconds, the page takes at least 3.6 seconds to load — before rendering.

Some visuals generate more than one query. A matrix with row and column groupings, conditional formatting, and a dynamic title might generate four or five queries. A card with a single measure generates one. Visual type and configuration determine query complexity.

The visual count rule

A practical guideline: aim for no more than **eight visuals per report page**. This is not a hard limit set by Power BI, but it is the point beyond which most reports start to feel sluggish.

FIG. 5.1 — VISUALS × TIME = USER PATIENCE



Use bookmarks, drill-through pages, and tooltips to split content — not more visuals on one page.

If you need to show more information, use bookmarks, drill-through pages, or tooltip pages instead of cramming more visuals onto one page. Important: hidden visuals still fire queries if visibility is controlled by a conditional expression.

Slicers — the hidden performance tax

Slicers look simple but can be surprisingly expensive:

- **High-cardinality slicers.** A slicer showing 50,000 unique values (customer names) has to load and render them all. Use a *dropdown* slicer instead of a list. Better yet, add a search box and reduce the displayed items.
- **Synced slicers.** When you sync a slicer across multiple pages, any selection triggers a query on all synced pages. Sync a year slicer across ten pages and changing the year fires queries on all ten — not just the visible one.
- **Multiple slicers.** Each slicer adds to filter context. Six slicers with cross-filtering enabled create complex combinations that multiply query cost.

Recommendations: limit slicers to three or four per page. Use dropdown mode for any slicer with more than 20 values. Avoid syncing slicers across more than three pages unless necessary. Consider the filter pane instead of visual slicers for less frequently used filters.

Cross-filtering & cross-highlighting

By default, clicking a data point cross-highlights the other visuals. This looks good but has a cost: every visual that reacts must re-query.

Go to *Format > Edit Interactions*. For each visual, choose: cross-highlight (default), cross-filter, or **none**. Setting a visual to "none" means it does not re-query when the user clicks other visuals. For static reference visuals (target cards, text boxes), this is free performance.

> *Cross-highlighting on three key visuals is helpful. Cross-highlighting on all twelve is just noise — and a tax on every click.*

Custom visuals

Custom visuals from AppSource can do things native visuals cannot. They can also create performance problems that are hard to diagnose.

Certified vs uncertified. Certified visuals have been reviewed by Microsoft for security and quality. Certification does not guarantee speed, but certified visuals are less likely to have memory leaks or excessive API calls.

External API calls. Some custom visuals call external services (map tiles, geocoding). These depend on network latency and the third party's availability — outside your control.

How to test: add the custom visual to a blank page with one measure. Record with Performance Analyser. Then put the same measure in a native card on another blank page. Compare. If the custom visual is much slower, decide whether the richness is worth it. If you can't replace it, isolate it on a drill-through page.

Report page load strategies

- **Landing page.** A simple navigation hub with no data visuals loads instantly. Data-heavy pages load only when chosen.
- **Progressive disclosure.** Show summary first (KPI cards, one chart). Buttons or tabs reveal more detail. Cuts initial query count.
- **Tooltip pages.** Move chart annotations and supplementary metrics to tooltip pages. They render on hover, not on page load.
- **Bookmark navigation.** Use bookmarks to create tabbed layouts. Only the active tab's visuals fire queries.

Performance Analyser walkthrough

Performance Analyser is built into Power BI Desktop. Here is how to use it:

1. Go to *View > Performance Analyser*. Click **Start Recording**.
2. Interact with your report. Click slicers, switch pages, expand a matrix. Each action generates trace data.
3. Stop the recording. You see a list of every visual with timing information.

The three metrics:

- **DAX query** — time spent executing the DAX. High → model or measure needs work (Chapters 2 & 3).
- **Visual display** — time rendering after data is returned. High → visual is complex or has too many data points. Reduce granularity.
- **Other** — everything else: network, security, query prep. High → check gateway, capacity, network.

> *If three visuals exceed 1,500 ms and seven are under 200 ms — fix those three. The rest are noise.*

Worked example. Page with ten visuals, 8-second total load. Three visuals exceed 1,500 ms; the other seven are under 200 ms. Focus on those three. Copy their queries and analyse in DAX Studio.

TRY THIS

1. Open Performance Analyser on your busiest report page. Record a full page load. Sort by DAX query time, longest first.
2. For the slowest visual: how many data points? Is the measure complex? Could it be simpler?
3. Try it: hide the three slowest visuals (select, Delete, then Ctrl+Z to undo). Re-record. How much faster?

CHAPTER SIX

06

DirectQuery & *Composite* Models.

Import is the fastest answer for most scenarios. But sometimes you cannot import — and that is where the trade-offs begin.

DIRECTQUERY | COMPOSITE | AGGREGATIONS | DUAL MODE

When to use DirectQuery

DirectQuery makes sense in three situations:

- **Real-time requirements.** The business needs data that is current to the minute or second. Even the fastest scheduled refresh (30 minutes on Premium) is not fresh enough.
- **Very large datasets.** The source has hundreds of millions or billions of rows. Importing would exceed memory limits and take hours to refresh.
- **Data governance.** Security policies require data stay in the source system. DirectQuery queries the source directly without storing data.

If none of these apply, use import mode. DirectQuery adds complexity and reduces performance in almost every case.

The cost of DirectQuery

Every visual interaction sends a query back to the source. There is no VertiPaq cache, no compression, no parallel storage-engine scans. Performance depends entirely on the source.

COST	WHAT IT MEANS
Latency	Every query includes network round-trip time. If the source is on-prem via a gateway, add gateway processing time.
No compression	Source must scan raw data. A 50 ms import query might take 2 seconds in DirectQuery.
Source load	50 concurrent users = 50 users' worth of queries on the source. Can impact other applications.
Limited DAX	Some DAX patterns that work well in import perform poorly when translated to the source's SQL.

DirectQuery best practices

If you must use DirectQuery, follow these practices to minimise the impact:

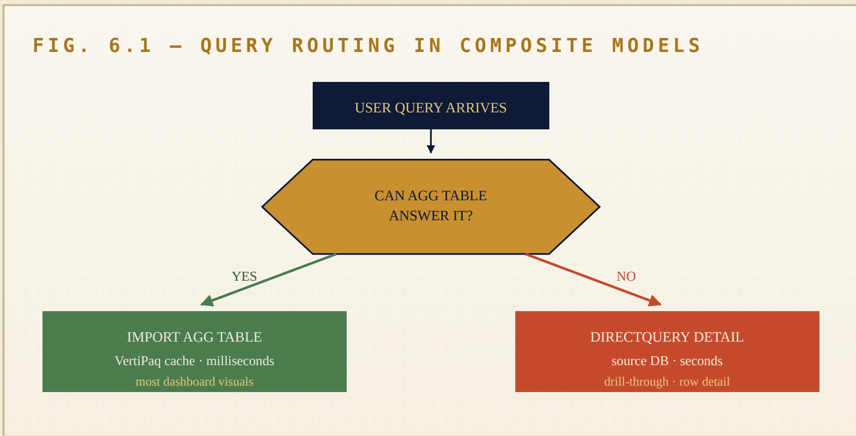
- **Create views on the source.** Don't point Power BI at raw tables. Create database views that pre-join, pre-filter, and pre-aggregate where possible. The view does the heavy lifting server-side.
- **Add indexes.** Ensure source tables have indexes on every column used in filters and relationships. Power BI generates `WHERE` clauses for slicer selections — without indexes, these become full table scans.
- **Limit visuals.** The visual-count rule from Chapter 5 is more critical with DQ. Every visual is a round-trip. Five or six visuals max per page.
- **Reduce cardinality in visuals.** A bar chart with 20 categories sends a simple `GROUP BY`. A scatter plot with 10,000 points sends a much heavier query.
- **Use aggregations.** Power BI user-defined aggregations let you create pre-aggregated import tables that satisfy most queries, with DQ fallback for detail. Best of both worlds — and the bridge to composite models.

> *The DirectQuery report that performs well is usually the one where the database engineer was at the table from day one.*

Composite models

A composite model combines import and DirectQuery tables in one model. The most common pattern is an imported aggregation layer with a DirectQuery detail layer.

FIG. 6.1 — QUERY ROUTING IN COMPOSITE MODELS



For example: import a `Sales_Aggregated` table with daily totals by product category (a few million rows), keep `Sales_Detail` (billions of rows) in DirectQuery. Most dashboard visuals show category-level or daily totals and hit the fast import table. When a user drills into individual transactions, the query falls back to DirectQuery.

Setting up aggregations · 6 steps

1. Create the aggregated table — a database view, a Power Query summary, or a source-defined table. Use the grain you need for most visuals (e.g., daily by product by region).
2. Import the aggregated table into Power BI.
3. Keep the detail table in DirectQuery mode.
4. On the aggregated table, set up *Manage Aggregations*. Map each aggregated column to its detail equivalent and specify the function (Sum, Count, Min, Max, GroupBy).
5. Hide the aggregated table from report view. Power BI uses it automatically.
6. Test. Performance Analyser + DAX Studio. Verify summary queries hit the import table and detail queries fall back to DirectQuery.

Dual storage mode

Dual mode sets a table to act as both import and DirectQuery. Power BI decides at query time which to use based on the other tables involved.

When to use dual. For dimension tables shared between import and DirectQuery fact tables. Avoids the performance penalty of cross-source joins.

When to avoid dual. For large fact tables. Dual on a fact table means the data is stored twice (VertiPaq *and* queried from source) — wastes memory without a clear benefit.

When the bottleneck is the database

One pattern shows up repeatedly when teams adopt DirectQuery for the first time: the source database becomes the limit, not Power BI. The visuals look slow, the obvious instinct is to optimise DAX or remove visuals, but neither moves the numbers. The real fix lives upstream — in indexes, in views, in the way the warehouse was modelled in the first place.

If your DirectQuery report is slow and you have already pruned visuals and tightened DAX, stop. Open DAX Studio, capture the SQL, and run it on the source. The numbers there will tell you whether your next move belongs in Power BI or in the database.

TRY THIS

1. If you have a DirectQuery report, open Performance Analyser and record a page load. Then open DAX Studio, enable Query Plan tracing, and run the same query. DAX Studio shows the SQL sent to the source.
2. Copy that SQL and run it directly on the source database. Compare. If the SQL is slow on the source, the fix is in the database — indexes, views, query tuning — not in Power BI.
3. For a large import model approaching size limits: do most visuals need row-level detail, or do they work with aggregated data? If aggregated data covers 80% of queries, a composite model could dramatically improve both performance and size.

CHAPTER SEVEN

07

Deployment, *Capacity* & Service.

A report that runs beautifully on your laptop can struggle in the Service. Different hardware, shared capacity, scheduled refreshes — the equation changes.

DESKTOP VS SERVICE | THROTTLING | REFRESH | RLS

Desktop vs Service performance differences

Power BI Desktop runs on your local machine with dedicated resources. The Power BI Service runs on shared infrastructure in the cloud. There are several important differences:

- **Query caching.** The Service caches query results. Multiple users with the same filter selections benefit from cached results. Desktop has no such caching.
- **Dataset size limits.** Desktop handles up to your machine's memory. The Service enforces licence-based limits (1 GB Pro, 10 GB PPU, configurable on Premium).
- **Concurrent load.** On Desktop you are the only user. In the Service, your dataset shares resources. During peak hours, query times rise.
- **Gateway overhead.** If your source is on-prem, every refresh and DirectQuery interaction goes through a data gateway. The gateway adds latency and can be a bottleneck.

If a report is noticeably slower in the Service than on Desktop, the cause is usually capacity pressure, gateway latency, or a dataset too large for the available memory.

> *If reports are slow at 9 AM and fast at 7 PM, your capacity is under pressure during business hours — not your DAX.*

Capacity planning

Power BI offers three main tiers, each handling performance differently:

FEATURE	PRO (SHARED)	PREMIUM PER USER	PREMIUM / FABRIC
Dedicated resources	No — shared pool	Per-user allocation	Yes — dedicated capacity
Max dataset	1 GB	10 GB	Configurable (25+ GB)
Query concurrency	Shared, throttled	Better, per-user	High, dedicated cores
Refresh / day	8	48	48 (or XMLA)
Throttling	Aggressive	Moderate	Based on SKU size

Throttling is the most common cause of unexplained slowness in the Service. When a capacity is overloaded, Power BI queues incoming requests and slows down query execution. On shared capacity (Pro), throttling happens frequently during business hours. On Premium, throttling depends on SKU size and total workload.

If users report "sometimes slow," check whether slowness correlates with time of day. If reports are slow at 9 AM and fast at 7 PM, the capacity is under pressure during business hours.

Scheduled refresh best practices

Scheduled refreshes compete with interactive queries for capacity. Three large datasets refreshing simultaneously while users are viewing reports = everything slows down.

- **Stagger refreshes.** Don't schedule everything at the same time. Spread across the day. Run large refreshes off-peak (early morning, late evening).
- **Gateway management.** An on-prem gateway is a single point of contention. If ten datasets refresh through the same gateway at 6 AM, they queue up. Stagger or add gateway nodes in a cluster.
- **Incremental refresh** for large datasets (Chapter 4) cuts load on both gateway and capacity by processing only changed partitions.
- **Monitor refresh duration.** Check refresh history regularly. If times trend upward, the dataset is growing — act before it hits the limit.

> *The refresh schedule is a load profile. Treat it like one: plot the times, balance the load, and never let three big jobs land in the same minute.*

If you cannot stagger refreshes (sometimes business rules force overlap), invest in gateway clustering instead. Two gateway nodes in a cluster halve the contention on cross-source merges and DirectQuery paths.

Fabric Capacity Metrics App

The Fabric Capacity Metrics App is a Microsoft-provided Power BI app that monitors your Premium or Fabric capacity usage. It is the primary tool for diagnosing capacity-level performance issues.

To install: go to AppSource in the Power BI Service and search for "*Microsoft Fabric Capacity Metrics.*" Install and connect it to your capacity.

KEY METRICS TO WATCH

- **CU (Capacity Unit) usage.** How much of your capacity's processing power is consumed. Sustained usage above 80% means you are approaching throttling.
- **Overload periods.** Time ranges where demand exceeded capacity. Queries were queued or rejected.
- **Per-item breakdown.** Which datasets, dataflows, and reports consume the most. This is where you find the dataset eating 60% of your capacity by itself.
- **Background vs interactive.** Distinguishes refreshes from user queries. Background dominant → stagger refreshes. Interactive dominant → optimise the reports.

When you find the problem dataset, go back to Chapters 2 and 3. Reduce model size, optimise DAX, re-check. The capacity metrics will confirm whether your changes had impact.

Deployment pipelines

Deployment pipelines provide a Dev / Test / Prod workflow for Power BI content. From a performance perspective, the key benefit is the ability to test performance *before* promoting to production.

- **Dev.** Build and iterate. Performance is not the primary concern here.
- **Test.** Connect to production-sized data (or a representative sample). Run performance tests with Performance Analyser and DAX Studio. Verify query times are acceptable.
- **Prod.** Only promote content that passes performance testing. Use deployment rules to swap connection strings between environments.

XMLA endpoints. Premium and Fabric capacities expose XMLA endpoints that allow external tools (Tabular Editor, DAX Studio, SQL Server Management Studio) to connect directly to the published dataset. Use these endpoints to run VertiPaq Analyser on the production dataset and verify model size and compression.

> *Performance regressions caught in Test cost minutes. Performance regressions caught in Prod cost trust.*

If you do not yet have a deployment pipeline, this is one of the highest-leverage governance changes you can make. The pipeline does not by itself make any report faster — but it stops slow reports reaching users.

Row-Level Security performance

RLS adds filters to every query based on the user's identity. This has a measurable performance impact.

- **Static RLS** uses hardcoded filter values (e.g., `Region = "EMEA"`). Simple filter, similar to a slicer. Performance impact small.
- **Dynamic RLS** uses `USERPRINCIPALNAME()` or similar. Engine evaluates the function, looks up permissions, applies filter. Adds overhead — especially with complex permission tables.

DYNAMIC RLS · RELATIVE COST

PATTERN	IMPACT
`USERPRINCIPALNAME() = SecurityTable[Email]`	**Low** — simple lookup
`PATHCONTAINS` on hierarchy security table	**Medium** — path traversal
Complex `CALCULATE` with multiple `FILTER` in RLS	**High** — evaluated every query
Dynamic RLS with bidirectional relationships	**High** — compounds bidirectional cost

Testing RLS. In Power BI Desktop, go to *Modeling > View as Roles*. Select a role and optionally enter a specific user's email. This simulates the query with RLS applied. Run Performance Analyser in this mode to see the actual impact.

For a thorough test, create two or three profiles: one with full access, one restricted to a single region, one restricted to a small team. Compare query times across profiles. If the restricted profile is significantly slower than full, the RLS expression needs optimisation.

Three signals you have a capacity problem

Most of the time, what looks like a slow report is really a capacity that is running too hot. Three patterns make that diagnosis easy:

- **Time-of-day correlation.** Reports are slow at 9 AM and fast at 7 PM. The DAX has not changed since lunchtime — but the contention has.
- **One dataset eats the budget.** The Capacity Metrics App shows a single dataset consuming a disproportionate share of CUs. Usually the same one users complain about.
- **Background dominates interactive.** Refreshes are queued so densely that user queries can't get a thread. Stagger the refreshes; the report gets faster without a single DAX change.

Performance work that ignores capacity is performance work that gets undone every Monday morning.

TRY THIS

1. If you have Premium/Fabric capacity, install the Capacity Metrics App and look at the last 7 days. When is CU usage highest? Does it correlate with refreshes, user activity, or both?
2. Find the single dataset consuming most CU. Open it in DAX Studio via XMLA. Run VertiPaq Analyser. Larger than it needs to be?
3. If you use RLS, test with View as Roles. Compare query times between a restricted user and full access. More than 30% difference = your RLS is a bottleneck.

CHAPTER EIGHT

08

The Performance *Optimisation* Playbook.

A repeatable process for auditing and improving any Power BI report. Measure. Identify. Fix. Re-measure. Then do it again.

AUDIT | SCORECARD | 15 KILLERS | TOOLS

The performance audit framework

Every performance optimisation follows the same four-step cycle. Skip a step and you risk wasting time on the wrong problem or making changes you cannot verify.

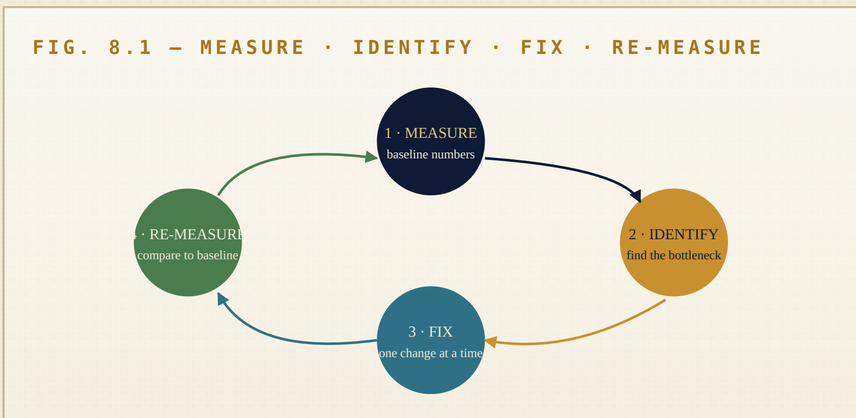


FIG. 8.1 — MEASURE · IDENTIFY · FIX · RE-MEASURE

- **1 · Measure.** Open Performance Analyser, record a full page load for every report page. In DAX Studio, capture Server Timings for the top five slowest measures. Record model size with VertiPaq Analyser. Document everything in a spreadsheet — you will need these numbers later.
- **2 · Identify.** Use the diagnostic checklist from Chapter 1 to narrow the category. Model size? DAX complexity? Visual count? Throttling? Drill into the relevant chapter.
- **3 · Fix.** Make one change at a time. Don't rewrite five measures and remove ten columns simultaneously. You will not know which change helped.
- **4 · Re-measure.** After each change, re-run the same measurements. Compare. If improved, keep. If not, revert and try something else.

The performance scorecard

Use this scorecard to assess any Power BI model. Check each item. Any unchecked item is a potential performance issue.

MODEL LAYER

- Star schema implemented (no snowflake chains)
- Auto date/time disabled
- Dedicated date table created and marked
- No unnecessary columns imported
- No high-cardinality text or GUID columns in fact tables
- DateTime columns truncated to Date where appropriate
- No bidirectional relationships unless required
- Model size within licence limits with headroom

DAX LAYER

- No SUMX / AVERAGEX when SUM / AVERAGE would suffice
- No nested CALCULATE without purpose
- Variables used for repeated measure references
- FILTER uses column predicates, not whole-table scans
- No calculated columns that should be measures
- Time intelligence uses marked date table
- No DISTINCTCOUNT on high-cardinality strings
- Top 3 slowest measures optimised (FE time < 500 ms)

VISUAL LAYER

- Max 8 visuals per page
- Max 4 slicers per page
- High-cardinality slicers use dropdown mode
- Cross-filtering disabled on static visuals
- Custom visuals tested for performance impact
- Drill-through used for detail instead of crowded pages

SERVICE LAYER

- Scheduled refreshes staggered (not all at same time)
- Gateway not saturated during refresh windows
- Capacity CU usage below 80% during peak hours
- Incremental refresh configured for large datasets
- RLS tested with restricted user profile
- Performance tested in Test stage before Prod promotion

> *Print this. Pin it next to your monitor. Run it on every model before publishing.*

Top 15 performance killers · quick reference

WHAT IT IS	HOW TO SPOT IT	HOW TO FIX IT
High-cardinality columns	VertiPaq Analyser: cardinality near row count	Remove, truncate, or bucket
Too many columns	Many small columns adding up	Remove unused columns in PQ
Auto date/time tables	Hidden date tables in model view	Disable in Options > Data Load
Snowflake schema	Chained dimension tables	Flatten into star schema
Bidirectional relationships	Double arrows in model view	Switch to unidirectional
SUMX on simple aggregation	DAX code review	Replace with SUM
Nested CALCULATE	DAX code review	Flatten to single CALCULATE
FILTER on whole table	DAX Studio: high SE query count	Use column predicate
No VAR caching	Repeated measure references	Store in VAR, reference from RETURN
Calc columns as measures	Calc columns in fact tables	Convert to measures
Too many visuals	Performance Analyser count	Reduce to 8 max, use bookmarks
High-card slicers	Inspection + Perf Analyser	Dropdown or search box
Broken query folding	"View Native Query" greyed out	Reorder steps
Simultaneous refreshes	Refresh history: overlapping	Stagger refreshes
Capacity throttling	Capacity Metrics: overload	Optimise top consumers or upsize

Tools reference

- **DAX Studio** (`daxstudio.org`) — free, open-source. Connect to any Power BI model. Run queries, capture Server Timings, run VertiPaq Analyser. Essential for diagnosing slow measures.
- **Tabular Editor** (`tabulareditor.com`) — external model editor. V2 free. Edit measures, tables, relationships outside Power BI Desktop. Best Practice Analyser automates model quality checks.
- **Performance Analyser** — built into Power BI Desktop. Per-visual timing. First step in every performance investigation.
- **Fabric Capacity Metrics App** — Microsoft-provided monitoring app. CU usage, overload periods, per-item consumption.
- **Best Practice Analyser** — part of Tabular Editor. Configurable ruleset against your model. Standalone ruleset at `github.com/TabularEditor/BestPracticeRules` .

Recommended learning path

Once you have applied the techniques in this book, here is where to go next:

- **For deeper DAX** — study Marco Russo and Alberto Ferrari at `sqlbi.com`. *The Definitive Guide to DAX* is the most comprehensive reference available.
- **For model design** — Microsoft's Power BI guidance documentation, particularly the star schema design guidance and performance optimisation whitepapers.
- **For advanced diagnostics** — learn to read DAX query plans in DAX Studio. Physical and logical plans give you insight into how the engine processes your measures.
- **For capacity management** — Microsoft Fabric docs on capacity, throttling, and smoothing. Rules evolve — stay current.
- **For community** — the Power BI Community forums, the SQLBI blog, and the Power BI subreddit are active sources of real-world performance discussions.

> *Performance is not a project. It is a habit. The teams that ship fast reports are the same teams every month — because they audit every month.*

TRY THIS

1. Print the Performance Scorecard from this chapter. Open your most important Power BI model. Work through every checklist item.
2. For each unchecked item, estimate effort (Low/Medium/High) and impact (Low/Medium/High). Start with High impact + Low effort — your quick wins.
3. After making changes, re-run the full audit. Compare model size, query times, and page load times against your baseline. Document improvements. Share with your team.

APPENDIX A

DAX Quick Reference.

A one-page cheat sheet of slow vs fast DAX patterns. Pin this to your wall.

PATTERN	SLOW	FAST
Simple aggregation	`SUMX(Sales, Sales[Amount])`	`SUM(Sales[Amount])`
Nested CALCULATE	`CALCULATE(CALCULATE(…))`	Single `CALCULATE` with filters
Blank check	`IF(ISBLANK([M]),0,[M])`	`COALESCE([M], 0)`
Table filter	`FILTER(Sales, Sales[Col]="X")`	`Sales[Col]="X"` in `CALCULATE`
Distinct count	`DISTINCTCOUNT(Sales[GUID])`	`DISTINCTCOUNT(Sales[IntID])`
Remove filters	`ALLEXCEPT(WideTable, Col)`	`REMOVEFILTERS(specific cols)`
Time intelligence	Unmarked date table	Marked date table
Repeated measures	Ref `[M]` 3× without VAR	`VAR _m = [M] RETURN …`
Ranking	`RANKX(ALL(BigTable), …)`	`ADDCOLUMNS + COUNTROWS`
Derived values	Calculated column on fact	Measure with `SUMX`

> *Most slow DAX is not exotic. It is the same ten patterns, written by ten different people, over and over again.*

APPENDIX B

Power Query Folding Cheat Sheet.

Folding behaviour when connected to SQL Server. Other sources may differ.

FOLDING-SAFE · RUNS ON SERVER	FOLDING-BREAKING · RUNS LOCALLY
`Table.SelectColumns` (Remove/Choose)	`Table.AddColumn` with custom M function
`Table.SelectRows` (Filter Rows)	`Table.Pivot` / `Table.Unpivot`
`Table.Sort` (Sort Rows)	`Table.Buffer`
`Table.RenameColumns` (Rename)	`Table.AddIndexColumn`
`Table.TransformColumnTypes` (Change Type)	`Table.Combine` (append from different source)
`Table.ReplaceValue` (simple)	Merges across different sources
`Table.FirstN` / `Table.LastN`	`Text.Combine` with custom logic
`Table.Group` (some sources)	`Table.FillDown` / `Table.FillUp`
`Table.NestedJoin` (same source)	Running totals / cumulative functions

KEY RULE

Folding-safe steps must come **before** folding-breaking steps. Once folding breaks, all subsequent steps run locally regardless of their type.

How to check folding: right-click any step in Applied Steps. If "View Native Query" is available and shows SQL, that step folds. If greyed out, folding is broken at or before that step.

Glossary.

Every term used in this book, in plain English.

VertiPaq. The in-memory columnar storage engine used by Power BI in Import mode. Compresses data using dictionary encoding and RLE, enabling fast scans and aggregations.

Cardinality. The number of unique values in a column. Low cardinality compresses well. High cardinality compresses poorly and inflates memory usage.

Query Folding. The ability of Power Query to translate M code steps into native source queries (e.g., SQL). When folding works, the source does the heavy lifting and only sends back what is needed.

SE — Storage Engine. The component of VertiPaq that scans compressed data and performs simple aggregations. Runs in parallel across multiple threads. Fast path for queries.

FE — Formula Engine. Component that handles everything SE cannot: complex calculations, row iteration, context transitions. Single-threaded. Usually the bottleneck in slow queries.

RLS — Row-Level Security. A feature that restricts data access at row level based on the user's identity. Dynamic RLS uses functions like `USERPRINCIPALNAME()` at query time.

Glossary *— continued*

CU — Capacity Unit. The unit of compute used by Microsoft Fabric and Power BI Premium to measure resource consumption. Each operation consumes CUs. Exceeding the allocation triggers throttling.

Composite Model. A Power BI model combining Import and DirectQuery storage modes. Typically uses imported aggregation tables for fast summary queries with DirectQuery fallback for detail.

Context Transition. The process by which a row context is converted into a filter context when `CALCULATE` or a measure reference is evaluated inside an iterator. Each transition has a cost.

Dictionary Encoding. A compression technique where each unique value is stored once in a dictionary, and rows store short references instead of full values. Effective when cardinality is low.

RLE — Run-Length Encoding. A compression technique where consecutive identical values are stored as a single value plus a repeat count. Effective when data is sorted and values repeat in sequence.

DirectQuery. A storage mode where Power BI sends every visual's query to the source database in real time, rather than importing the data. Higher latency, no compression — but always fresh.

Throttling. The slowdown applied when a Power BI capacity exceeds its CU allocation. Queries queue, refreshes delay, users wait. The most common cause of unexplained Service slowness.

∴

A Final Word.

From the author.

Thank you for reading *The Power BI Performance Bible*. I hope it saves you time, frustration, and a few Tuesday mornings.

Performance is rarely about a single magical fix. It is about a habit — looking at the model first, the DAX second, the visuals third, and the Service last. Following the same audit every time. Making one change at a time. Re-measuring. Writing down what worked.

If there is one idea I would like you to take from this book, it is this: **most slow Power BI reports can be made fast in less than a day.** You don't need to rebuild from scratch. You need to know where to look, what to change, and how to prove it worked.

If you found this book useful, share it with a colleague who is staring at a slow report right now. They will thank you. Probably with coffee.

— *Syed Hussnain Tahir Sherazi*

Leicester, 2025

✦

MICROSOFT POWER BI · PERFORMANCE & OPTIMISATION

Slow reports kill adoption. This book fixes that.

You can spend weeks gathering requirements, modelling data, and writing DAX — and lose all of it the moment a user decides your report is too slow to use. Performance is not a technical nice-to-have. Performance is a trust issue.

Inside this field manual you will find the exact patterns, anti-patterns, diagnostic steps, and fixes that turn forty-five-second reports into two-second reports. Real DAX. Real model changes. Real numbers.

> *"Most slow Power BI reports can be made fast in less than a day. You just need to know where to look — and what to change."*

Pragmatic, opinionated and refreshingly free of theory for theory's sake. The DAX anti-pattern chapter alone is worth the price.

— BI PRACTITIONER QUARTERLY

Reads like sitting next to a senior consultant who actually fixes the problem in front of you. The Try-This sections are gold.

— PUBLIC-SECTOR DATA REVIEW

Syed Hussnain Tahir Sherazi

DATA ENGINEER · LEICESTER · 2026

contact@syedhussnain.com
linkedin.com/in/hussnainsherazi

ISBN 978-1-0000-0001-0
£34.99 · $44.99 · €39.99