AI-Enhanced Programming Reference

Power Fx, Power Query (M), and DAX

A comparative technical handbook for R, Python, and PostgreSQL programmers

Edwin Alvarado-Mena, PhD

2026-11-27

Table of contents

1 Orientation: three languages, three responsibilities **4**
- 1.1 Why separate languages? . . . 4
- 1.2 A practical mental model . . . 4

2 Language foundations and execution **6**
- 2.1 Paradigms and evaluation . . . 6
- 2.2 Values, types and absence . . . 7
- 2.3 Execution boundaries and optimization . . . 8
- 2.4 A closer look at execution and language semantics . . . 9

3 Practice model and conventions **11**
- 3.1 Environment setup . . . 11

4 Comparative coding drills **13**
- 4.1 Coding conventions used in this handbook . . . 13
- 4.2 Reading comments across languages . . . 13
- 4.3 Scalar bindings and arithmetic . . . 14
- 4.4 Conditional logic . . . 15
- 4.5 String slicing . . . 16
- 4.6 String search and regex . . . 16
- 4.7 Dates and intervals . . . 17
- 4.8 Type conversion . . . 18
- 4.9 Missing values . . . 18
- 4.10 Error handling . . . 19
- 4.11 Filter rows . . . 20
- 4.12 Projection . . . 21
- 4.13 Create a column . . . 21
- 4.14 Sort and take top N . . . 22
- 4.15 Unique values . . . 23
- 4.16 Aggregate an expression . . . 24
- 4.17 Grouped aggregation . . . 25
- 4.18 Many-to-one left lookup . . . 25
- 4.19 Append tables . . . 26
- 4.20 Set intersection . . . 27
- 4.21 Nested values . . . 28
- 4.22 Reshape wide . . . 29
- 4.23 Map and reduce . . . 30
- 4.24 Reusable function . . . 31
- 4.25 Ranking . . . 32
- 4.26 Descriptive statistics . . . 33
- 4.27 API ingestion . . . 34
- 4.28 State and durable writes . . . 34
- 4.29 Compose a transformation pipeline . . . 35
- 4.30 Running revenue . . . 37
- 4.31 Reshape long . . . 38

5 Power Fx: application behavior and reactive formulas **40**
- 5.1 The Excel analogy and its limits . . . 40
- 5.2 Record scopes, nested expressions and validation . . . 41
- 5.3 Delegation lab . . . 42

6 Power Query (M): a functional integration language **43**
- 6.1 A complete pipeline . . . 43
- 6.2 Ingestion, nested values and reusable transforms . . . 44

6.3 ETL design and source pushdown . . . 45
6.4 Query folding: from an M expression to a source query . . . 45

7 DAX: semantic models and contextual analytics 48
7.1 Power BI as a complete analytical system . . . 48
7.2 What a semantic model actually contains . . . 50
7.3 Storage modes and the two analytical engines . . . 51
7.4 Build a valid star schema . . . 52
7.5 Measures, calculated columns and queries . . . 52
7.6 DAX evaluation contexts and context transition . . . 53
7.7 `CALCULATE()` and filter manipulation . . . 55
7.8 Virtual tables, lineage and relationships . . . 58
7.9 Ratios, time and non-additive measures . . . 59
7.10 DAX query view, ranking and statistics . . . 61
7.11 Performance practice . . . 62

8 Architecture and engineering decisions 63
8.1 Choose the right execution layer . . . 63
8.2 Power Apps versus conventional application frameworks . . . 63
8.3 Engineering lifecycle comparison . . . 64

9 Integrated case study: AI-assisted vocabulary learning 65
9.1 Requirements and data model . . . 65
9.2 Microsoft-oriented architecture . . . 65
9.3 Illustrative measures and analytical decisions . . . 66

10 Technical learning, adoption and mastery 68
10.1 Mapping transferable expertise . . . 68
10.2 Principles for technical learning and adoption . . . 68
10.3 Core questions and answers . . . 69
10.4 Extended questions and answers . . . 69
10.5 Training demonstration design . . . 72
10.6 Misconceptions to address . . . 72

11 Reference sheets 74
11.1 Function families . . . 74
11.2 Glossary . . . 75

12 Sources and further reading 76

1 Orientation: three languages, three responsibilities

The most productive way to approach the Microsoft stack is to separate application behavior, data preparation, and analytical semantics. The three languages overlap in their ability to manipulate tables, but the same expression has different consequences depending on where it runs. This handbook assumes familiarity with relational algebra, vectorized transformations, functional composition and software engineering. It emphasizes new execution models rather than elementary syntax.

Language	Primary abstraction	Typical execution boundary	Best reason to use it
Power Fx	Reactive formulas, records, tables	App host and connected service	Interactive business applications
Power Query (M)	Immutable expressions and data mashups	Query engine and source	Repeatable ingestion and shaping
DAX	Semantic model, evaluation context	Analytical engine	Interactive measures across dimensions
PostgreSQL	Relations and database state	Database server	Transactional integrity and set-based computation
R	Vectors, data frames, functions	R runtime or translated backend	Statistics and reproducible analysis
Python	Objects, functions, libraries	Python runtime and native extensions	General-purpose engineering and AI

The architecture is not a ranking of languages. PostgreSQL can calculate the same revenue aggregate as DAX, but DAX binds that calculation to report filters and established model relationships. Python can implement an ETL pipeline, but M exposes the pipeline to a graphical transformation editor. R Shiny can implement a reactive application, but Power Fx embeds reactivity inside a managed low-code host. These are differences in packaging, semantics and operational boundary, not mathematical expressiveness alone.

1.1 Why separate languages?

Power Fx derives from the spreadsheet-style formula model and emphasizes automatic recalculation alongside explicitly triggered behavior. M was designed to compose repeatable data mashups from heterogeneous sources. DAX addresses calculations over tabular semantic models. These documented design purposes support an architectural interpretation: Microsoft chose specialized abstractions rather than a single general-purpose language. This was not a technical necessity. SQL engines can expose files and JSON; Python and R can implement reactive interfaces and analytical systems. The benefit of specialization is integration with managed tools, and the cost is fragmentation across runtimes, syntax, governance and deployment. See the language specifications and references in the bibliography [1–6].

1.2 A practical mental model

Layer	Microsoft approach	Familiar analogy	Boundary to respect
Storage	Dataverse or relational source	PostgreSQL tables and constraints	A UI formula is not a database constraint
Application	Power Apps / Power Fx	Shiny, Streamlit, web app	App state is not durable state
Integration	Power Query / M	dplyr, pandas, ETL pipelines	Refresh is not a transaction
Semantic model	Power BI relationships	Star schema plus metrics layer	Relationships govern filter propagation

Layer	Microsoft approach	Familiar analogy	Boundary to respect
Analytics	DAX measures	Parameterized SQL aggregations	Context is supplied by the report
Orchestration	Power Automate	Scheduler or workflow service	Retries and identity need design

A semantic model is a governed description of business entities, relationships and calculations. It centralizes definitions that otherwise get duplicated across SQL queries or analysis notebooks. In Power BI, a visual issues a query against this model; it does not simply display a static spreadsheet. Treat the model as a software artifact, with tests for metrics and relationships.

Takeaway. Transfer your understanding of joins, data types, functions and aggregation, but learn the execution boundary of each language before optimizing or designing production solutions.

2 Language foundations and execution

2.1 Paradigms and evaluation

Dimension	Power Fx	M	DAX
Paradigm	Declarative formulas; imperative behaviors	Functional expressions	Declarative analytical expressions
State	App variables and data writes in behavior formulas	Immutable values	No mutable measure state
Evaluation	Dependency recalculation or user event	Dependency-driven; some expressions lazy	Context-dependent query evaluation
Types	Static checking and inference, host-specific types	Value types, annotations and runtime checks	Scalar types and tables; coercion
Abstraction	Named formulas, host-supported UDFs	First-class functions	Measures, variables, host-dependent UDF support
Scope	Record scope, local `With()`, app/screen variables	Lexical scope, `let` bindings	Row and filter context, lexical `VAR`
Recursion	Recursive canvas-app UDFs not supported	Recursive functions with `@`	Iterators and UDFs; recursive UDFs not supported
OOP	No conventional class hierarchy	No conventional OOP	No conventional OOP

In Power Fx, a label whose `Text` property references a slider is recalculated when the slider changes. The corresponding Shiny analogy is a reactive expression, not a loop waiting for a new value. Behavior formulas such as a button's `OnSelect` are different: they execute actions, including writes, when triggered.

M uses dependencies rather than requiring evaluation in textual order. List, record, table member expressions and `let` bindings are evaluated lazily when needed; other expressions may be evaluated eagerly. Laziness does not imply that an entire data source is never read, or that successive refreshes share a cache. R also uses lazy function arguments, but its promises and side effects differ from M's model.

DAX context is not just lexical scope. A measure can refer to a column in another table through relationships, and the same measure can return a different scalar for each cell of a matrix. `VAR` binds an expression's result in its defining context; it is not a mutable variable or a deferred instruction to recalculate after `CALCULATE()`.

Dimension	PostgreSQL	R	Python
Paradigm	Declarative SQL, procedural PL/pgSQL	Functional and imperative; OOP systems	Imperative, functional, object-oriented
State	Transactional tables and session state	Environments and copy-on-modify values	Mutable and immutable objects
Types	Declared SQL types and casts	Dynamic; vector coercion	Dynamic; optional type annotations
Evaluation	Optimizer chooses physical plan	Lazy arguments, vectorized operations	Generally eager; generators lazy
Abstraction	Functions, views, CTEs	Closures, packages	Functions, classes, modules
Scope	Query aliases, correlated subqueries	Lexical scope	Lexical scope; local/`global`/`nonlocal`

PostgreSQL's optimizer may reorder operations or inline CTEs. Do not infer physical execution order from SQL text. Similarly, pandas operations may run in native code despite Python's interpreted runtime. Distinguish semantics from implementation details when comparing performance.

2.2 Values, types and absence

Concept	Power Fx	M	DAX
Table	Table of records	Table with schema	Model or virtual table
Record	Named fields	Named fields	No general record value
List	Single-column table often used	Native list	One-column table constructor
Missing	`Blank()`	`null`	`BLANK()`
Error	Error value; `IfError()`	Evaluation error; `try` exposes error details	Error or blank, depending on operation
Equality	`=`, generally case insensitive for text	`=`, case-sensitive text	`=` coercive; `==` strict

M `null` equality differs from SQL's three-valued logic: `null = null` is true in M, while SQL's `NULL = NULL` is unknown. DAX blank is not equivalent to SQL `NULL`; loose equality can treat blank like zero or an empty string. Power Fx `IsBlank()` also recognizes empty text. In R, `NA` differs from `NULL` and `NaN`; pandas can represent missing values with `pd.NA`, `NaN`, `NaT` or `None`. Explicitly decide whether absence means zero, unknown, not applicable or no rows.

Notes:

- DAX strict equality (`==`) primarily differs from `=` by preventing `BLANK` coercions. It is not a case-sensitive text-comparison operator; use `EXACT()` when case-sensitive text comparison is required.
- Text comparisons are often case-insensitive in canvas-app scenarios, but behavior can depend on the specific function, connector, data source, and collation settings.

Concept	PostgreSQL	R	Python
Nested values	Arrays, composite types, `jsonb`	Lists, list columns	Lists, dicts, nested objects
Missing	`NULL`	Typed `NA`, `NaN`, `NULL`	`None`, pandas `NA`/`NaN`/`NaT`
Error	Statement exception; transaction impact	Conditions, `tryCatch()`	Exceptions, `try`/`except`
Equality	`=`, `NULL` yields unknown	`==`, `NA` propagates	`==`; `None` identity with `is`
Indexing	SQL string offsets usually 1-based	1-based vectors	0-based sequences

PostgreSQL arrays can have nonstandard lower bounds, although this is an advanced capability that is rarely encountered in typical application development. M uses zero-based positional access, while common DAX and Power Fx text slicing starts at one. Numeric representations also differ: PostgreSQL `numeric` supports exact decimal arithmetic where representable; R and Python floats usually use binary floating point; DAX supports whole numbers, fixed decimal and floating decimal types. Currency comparisons and ranking can be affected by rounding. Set precision explicitly instead of assuming identical numeric behavior.

Language	Numeric representation and precision boundary
Power Fx	Canvas apps use binary Float for Number; other hosts may use Decimal
M	Decimal Number is binary floating point; Whole Number and Fixed Decimal Number use distinct representations in Power Query

Language	Numeric representation and precision boundary
DAX	Whole Number is signed 64-bit; Decimal Number is floating point; Fixed Decimal Number has four fractional digits
R	Integer and double vectors; integer arithmetic can overflow to NA
Python	Arbitrary-precision int; binary float; decimal.Decimal with a configurable arithmetic context
PostgreSQL	Bounded integer types, numeric/decimal, real and double precision

Representation, conversion and arithmetic precision are different contracts. For example, binary floating point cannot exactly represent 0.1; fixed decimal may round values to its scale; integer conversion can overflow or reject fractional input. Choose explicit conversions and test boundary values, including sums near the representable limit. A decimal storage type does not guarantee that every intermediate operation uses decimal arithmetic [20–22].

Numeric representation in Power Fx depends partly on host support and data source interaction. Canvas apps historically use floating-point Number values, while other hosts and scenarios can expose decimal behavior.

2.3 Execution boundaries and optimization

Mechanism	Where computation runs	Failure or cost to watch
Power Fx delegation	Connector translates supported expression to remote source	Nondelegable operation can return only an initial subset
M query folding	Source executes translated transformations	Unfolded work increases transfer and refresh cost
DAX storage engine	VertiPaq or DirectQuery storage-engine requests to an external source	High-cardinality scans and excessive engine callbacks
PostgreSQL planner	Database server	Bad estimates, scans, spills, unsuitable indexes
dbplyr (R)	SQL backend through translated lazy expressions	Unsupported translation; collect downloads data
Python SQL libraries	Database if query is expressed and submitted there	Fetching into pandas moves computation locally

Delegation is a correctness issue, not just a performance issue. In canvas apps, nondelegable queries are subject to a configurable local row limit (Power Apps currently defaults to a delegation limit of 500 records and supports higher configured limits documented by Microsoft; verify current limits in the target environment). The exact supported operations depend on the connector and expression. A formula that works on ten rows may silently miss matches on a large source. Test against data beyond the limit, inspect warnings, and design delegable predicates [2].

Folding can be full, partial or absent. M's engine may translate a filter into SQL and then execute a custom transformation locally. Privacy boundaries can prevent otherwise possible combinations. `Table.Buffer()` is shallow buffering and may prevent downstream folding; it is not a universal speed optimization [3,4]. Diagnose using query plans, native-query inspection where available, and actual refresh timings.

Delegation and folding depend jointly on the expression, host, connector, source capabilities and configuration; neither is a guarantee of the language alone.

DAX's Formula Engine coordinates evaluation and delegates supported scans and aggregations to the Storage Engine. In an Import model, VertiPaq stores compressed columns, with cardinality affecting compression. DirectQuery sends requests to external sources. Avoid the simplistic rule that every iterator is slow: the expression, cardinality, engine pushdown and query plan determine actual cost.

Power BI storage approach	Where data resides	Practical consequence
Import	Compressed model cache	Fast interactive queries; freshness depends on refresh
DirectQuery	External source	Source latency, concurrency and feature restrictions matter
Dual	Import cache or DirectQuery depending on query	Useful in composite models; requires careful design
Direct Lake	Fabric/OneLake data with analytical engine access	Capacity, model configuration and fallback behavior matter

Storage mode is a deployment decision, not a language property. Import typically evaluates calculated columns and tables during refresh and measures on demand. DirectQuery restrictions are function- and context-specific, especially for calculated columns and RLS. Direct Lake features evolve; verify current capacity and model documentation before recommending it [7]. Parallelism and memory management are mostly engine-controlled in the Microsoft languages, whereas R and Python offer explicit library-based distributed and parallel computing. PostgreSQL may use parallel plans subject to planner and resource settings.

Takeaway. Optimize at the boundary: preserve correctness, minimize transfers, test actual plans, and avoid treating source pushdown as guaranteed.

2.4 A closer look at execution and language semantics

A type is a set of permitted values and operations, not merely a label for a column. PostgreSQL uses declared column types to enforce a storage contract. Python names can be rebound to objects of different types; a type annotation normally helps a checker rather than enforcing a runtime contract. R vectors usually coerce their elements to a common type. M values carry types, and table transformations can impose conversion rules. Power Fx checks formulas against the types exposed by a host and connector. DAX applies conversion rules while evaluating expressions over model columns. These differences matter when, for example, text containing a currency value is accepted in one runtime but rejected or rounded in another. Test boundary values instead of relying on implicit conversion.

A declarative expression describes a result. It leaves the choice of implementation to an engine. A reactive expression adds a dependency relationship: when an input changes, a dependent result can be recalculated. Imperative behavior describes actions, such as saving a record, in response to a user event. These categories are not mutually exclusive properties of entire languages. SQL is primarily declarative, but PL/pgSQL supports procedural code; Power Fx combines declarative properties with actions. R and Python can use libraries that implement declarative or reactive behavior without changing their core language semantics.

Lazy evaluation means deferring evaluation until a result is demanded. It does not mean running asynchronously or processing every operation in parallel. Consider an M expression with bindings for a source, a filtered result and an unused diagnostic branch. The engine need not evaluate the unused binding simply because it appears above the final expression. But producing the filtered result may require reading the entire source if no index or source translation can satisfy the predicate. In R, an unevaluated function argument is represented by a promise; in Python, a generator produces values as it is consumed. These are useful comparisons, not identical implementations.

Immutability and persistence are separate questions. M transformations return new values, and a logical intermediate table does not necessarily correspond to a separately allocated physical copy. PostgreSQL `UPDATE` changes transactional database state; successful commit and the configured durability guarantees determine persistence. A Python assignment changes a binding or, in the case of item assignment, may mutate an object. Power Fx behavior formulas can change application state or request a durable write. DAX measure evaluation does neither. When designing a system, ask whether an operation changes an underlying record, changes only a local object, or merely computes a new result.

M lists, tables and binary values can be streamed: repeated enumeration can trigger renewed evaluation or source access and may observe changed external data. An immutable binding is not a materialized snapshot or a transaction isolation guarantee. Buffering can stabilize a particular enumeration within an evaluation, but it is not a cross-refresh cache or source transaction [23].

Missing data requires a business policy. An absent sale, an unknown price and a legitimate zero-price sale have different meanings even if a visualization displays them similarly. A PostgreSQL aggregate usually ignores null inputs, whereas arithmetic involving `NULL` can propagate unknown values. An empty result in DAX is often blank; a zero result can signify a real observation. Replacing every blank with zero may generate misleading averages or make a report display categories that had no observations. The intended semantics should be specified before selecting a function.

In execution planning, distinguish a logical operation from its physical implementation. A filter may become an index scan, sequential scan or a predicate evaluated after another operation. PostgreSQL chooses among plans using statistics and costs; Power Query chooses a translation boundary and can ask PostgreSQL to optimize the translated query. Power Fx delegates supported expressions subject to connector limitations. A DAX query can combine Formula Engine work and Storage Engine requests. The fact that all these systems accept a filter expression does not establish where or how that expression runs.

3 Practice model and conventions

Use the included CSVs as the common dataset. Sales contains one row per order; all identifiers and dates are synthetic. Products and Customers are dimensions. Create a many-to-one relationship from Sales to each dimension in Power BI, with filtering from dimensions to Sales. Use a complete Calendar dimension with unique daily dates from 2025-01-01 to 2026-12-31, mark it as a date table for classic time intelligence, and connect Calendar[Date] to Sales[OrderDate].

OrderID	OrderDate	ProductID	CustomerID	Quantity	UnitPrice
1	2025-01-10	P1	C1	2	100
2	2025-02-15	P2	C2	3	50
3	2025-03-20	P3	C1	1	200
4	2025-04-10	P1	C3	4	100
5	2026-01-10	P1	C1	5	100
6	2026-02-15	P2	C2	2	50
7	2026-03-20	P3	C3	3	200
8	2026-04-10	P2	C1	4	50
9	2026-05-15	P1	C2	2	100
10	2026-06-20	P3	C3	2	200

Products: P1 Keyboard (Accessories, cost 60), P2 Mouse (Accessories, cost 30), P3 Monitor (Electronics, cost 120). Customers: C1 Alice (Costa Rica), C2 Bob (United States), C3 Carlos (Spain). These tables are in the project data directory. Prices are constant within a product for this exercise only.

3.1 Environment setup

In Power Apps, use local collections named Sales and Products and Customers (or connect typed data sources with those names); the formulas here assume local data unless specified. Import the files in Power Query and name the queries Sales, Products, Customers. Set types explicitly. Add Revenue as Quantity times UnitPrice to the Sales query or model, but not both under the same name. In DAX, create the following measure and calculated column in the Sales model if Revenue does not already exist.

DAX

```
Revenue =
// Calculated column: evaluate quantity times price for each fact row
Sales[Quantity] * Sales[UnitPrice]
```

DAX

```
Total Revenue =
// Measure: aggregate revenue in the current filter context
SUM (Sales[Revenue])
```

For the examples in R and Python, load the CSVs with their original case-sensitive column names. The project includes setup scripts for R, Python and PostgreSQL. PostgreSQL identifiers are intentionally unquoted and folded to lowercase; example queries use readable capitalized spelling without quotes. In all six environments the derived Revenue field is available after setup. Use English formula separators. DAX table expressions are intended for DAX Query View with EVALUATE, unless labeled as calculated tables; scalar formulas are measures or calculated columns as indicated. Power Fx examples use current canvas-app-style formulas and assume local collections for correctness demonstrations.

The Python examples use pandas as pd, NumPy as np, and standard library modules where indicated. R uses dplyr, tidyr, purrr, stringr and lubridate. Package methods are not native language features. PostgreSQL examples target commonly available PostgreSQL 16+ features, without extensions unless specified.

For a standalone scalar DAX drill, wrap the expression in `EVALUATE ROW("Result", expression)`; for a table drill, use `EVALUATE` followed by the table expression and an outer `ORDER BY` when ordering is required. A named measure or calculated column is instead entered in the corresponding model editor.

Record the host build and model compatibility level for DAX, the Power Apps host and connector, the Power Query host, the R and Python package versions, and the PostgreSQL server version. The common fixture assumes unique nonmissing dimension keys, complete numeric values and the stated date grain. Compare output schemas, types, duplicate policy, null policy and ordering explicitly; equal displayed numbers alone do not establish equivalent semantics.

Verification. The included Python fixture checker independently validates expected outputs, not the Microsoft runtimes. Test snippets in their actual host before production use.

Environment	Additional runtime verification required
Power Fx	Canvas formula compilation, local collections, delegation and connector errors
M	Query evaluation, types, deferred errors and source folding
DAX	Query View, model measures, filters, relationships and compatibility
R	Execution with recorded package versions
Python	Execution with recorded dependency versions
PostgreSQL	Execution against the declared server and schema

Test empty inputs, missing values, ties, duplicate or unmatched keys, case variants, precision boundaries and conflicting filters. Record actual execution separately from static review and fixture arithmetic.

4 Comparative coding drills

4.1 Coding conventions used in this handbook

Microsoft calls M the *Power Query M formula language*, also known as M. This handbook uses **Power Query (M)** for language labels, M for the language, and Power Query for the broader preparation technology and editor. See [17].

Language	Formatting convention	Reference
Power Fx	Descriptive names, indented nested calls, explicit record scopes	Microsoft coding guidelines
Power Query (M)	Four-space indentation, readable named steps, explicit types where useful	Microsoft language reference and formatter
DAX	Uppercase functions, qualified columns, unqualified measures, space before function parentheses	SQLBI DAX style
R	Two-space indentation, native pipes, tidyverse transformations	Tidyverse style
Python	Four-space indentation, vectorized pandas/NumPy, method chains	PEP 8 and PEP 257
PostgreSQL	Uppercase clauses, explicit joins and deterministic ordering	SQLFluff PostgreSQL conventions

These conventions are choices, not grammar requirements. Shared fixture names remain unchanged even where local naming conventions would prefer snake_case. R's native pipe `|>` passes a value into the next call; pandas method chains invoke a method on each intermediate object. `DataFrame.pipe()` allows an arbitrary function to participate in that chain. Neither notation guarantees lazy execution or remote computation. Use comments to explain intermediate logic, not to imply that a visual step is necessarily a physical execution step. Formatters and linters support readability but cannot establish correctness; use fixture-based tests alongside them.

4.2 Reading comments across languages

Comments document intent, assumptions and execution boundaries. They are ignored by the language parser; they are not a substitute for tests. Python docstrings are string literals with documentation semantics, not block comments. R and Python conventionally use repeated single-line comments for a longer explanation. PostgreSQL block comments may nest; do not assume the same nesting rules in other languages.

Language	Single-line comment	Block comment
Power Fx	`//`	`/* ... */`
Power Query (M)	`//`	`/* ... */`
DAX	`//` (also `--`)	`/* ... */`
R	`#`	No native block comment
Python	`#`	No native block comment
PostgreSQL	`--`	`/* ... */`

See the Power Fx grammar, M lexical specification, DAX comment conventions, R language definition, Python lexical reference, and PostgreSQL lexical reference.

The following examples illustrate the actual delimiters in context. Keep comments near the operation they describe and avoid commenting out part of a quoted literal.

Power Fx

```
// Single-line explanation
/* A longer explanation
   can span lines */
1 + 2
```

Power Query (M)

```
// Single-line explanation
/* A longer explanation
   can span lines */
1 + 2
```

DAX

```
// Single-line explanation
-- Alternative single-line delimiter
/* A longer explanation
   can span lines */
1 + 2
```

R

```
# Single-line explanation
# Continue on another line
1 + 2
```

Python

```
# Single-line explanation
# Continue on another line
1 + 2
```

PostgreSQL

```
-- Single-line explanation
/* A longer explanation
   can span lines */
SELECT 1 + 2;
```

Each exercise compares the same intention across six languages. Table-valued DAX expressions are not automatically measures; wrap them in `EVALUATE` in Query View. Expressions that return different types (scalar versus one-row SQL result, for example) are labeled where relevant. The drills favor transparent semantics over artificially identical syntax.

4.3 Scalar bindings and arithmetic

Calculate a 10% surcharge on 100; result 110.

Power Fx

```
// With binds a local record; no app state is modified
With({x: 100}, x * 1.1)
```

Power Query (M)

```
// Bind an immutable scalar, then return the expression
let
    x = 100
in
    x * 1.1
```

DAX

```
VAR x =
// VAR holds a scalar in the current context
100
// Evaluate the final expression using the bound variables
RETURN
    x * 1.1
```

R

```
# Bind a numeric vector of length one
x <- 100
x * 1.1
```

Python

```
# Bind a local name before evaluating the surcharge
x = 100
x * 1.1
```

PostgreSQL

```
-- The CTE exposes x as a column of a one-row relation
WITH v AS (
    SELECT 100 AS x
)
SELECT x * 1.1
FROM v;
```

The SQL CTE establishes a relation, not a scalar variable. M and DAX bindings are immutable; Power Fx `With()` is a local record binding. Floating-point representation can differ even when the displayed value is identical.

4.4 Conditional logic

Classify `200` as `"High"` if at least `300`, otherwise `"Standard"`.

Power Fx

```
// If returns one of two text values
If(200 >= 300, "High", "Standard")
```

Power Query (M)

```
// The if expression requires both branches
if 200 >= 300 then "High" else "Standard"
```

DAX

```
// IF returns a scalar in the current context
IF (200 >= 300, "High", "Standard")
```

R

```r
# ifelse is vectorized over its test
ifelse(200 >= 300, "High", "Standard")
```

Python

```python
# The conditional expression evaluates one branch
("High" if 200 >= 300 else "Standard")
```

PostgreSQL

```sql
-- CASE returns a value, rather than controlling statement flow
SELECT CASE WHEN 200 >= 300 THEN 'High' ELSE 'Standard' END;
```

Expected `"Standard"`. SQL `CASE` and R `ifelse()` are expressions; Python `if` as a statement is distinct from its conditional expression. Boolean short-circuit and missing-value behavior are not universally interchangeable.

4.5 String slicing

Extract characters 2–4 from `"Keyboard"`; result `"eyb"`.

Power Fx

```
// Mid uses a one-based start and a character count
Mid("Keyboard", 2, 3)
```

Power Query (M)

```
// Text.Middle uses a zero-based offset and count
Text.Middle("Keyboard", 1, 3)
```

DAX

```
// MID uses a one-based start and length
MID ("Keyboard", 2, 3)
```

R

```r
# substr uses inclusive start and end positions
substr("Keyboard", 2, 4)
```

Python

```python
# Slice from zero-based index 1 to exclusive index 4
("Keyboard"[1:4])
```

PostgreSQL

```sql
-- FROM is one-based; FOR specifies length
SELECT substring('Keyboard' FROM 2 FOR 3);
```

M and Python use zero-based offsets; R `substr()` uses inclusive endpoints. SQL and DAX use start and length. Unicode grapheme boundaries may not match code points or UTF-16 units.

4.6 String search and regex

Test whether `"Keyboard"` begins with `"Key"`; result true.

Power Fx

```
// StartsWith tests a prefix; it is not a regex
StartsWith("Keyboard", "Key")
```

Power Query (M)

```
// Test the literal prefix with the default comparer
Text.StartsWith("Keyboard", "Key")
```

DAX

```
// Compare a fixed-length prefix; not a regex
LEFT ("Keyboard", 3) == "Key"
```

R

```
# The anchor ^ matches the beginning of the string
str_detect("Keyboard", "^Key")
```

Python

```
# re.search returns a match object or None
re.search(r"^Key", "Keyboard") is not None
```

PostgreSQL

```
-- The tilde operator uses a PostgreSQL regex
SELECT 'Keyboard' ~ '^Key';
```

Power Fx `IsMatch()` supports regex patterns, with host-specific constraints. PostgreSQL uses POSIX-style regex, R `stringr` uses ICU, and Python `re` has its own dialect. M and conventional DAX have no general native regex facility in their core function set; prefix matching is enough here.

The examples agree for the supplied capitalization, not for arbitrary text. Power Fx StartsWith and typical DAX text comparisons can be case-insensitive; M Text.StartsWith defaults to a case-sensitive comparison, and the R, Python and PostgreSQL regex examples are case-sensitive by default. For DAX use `EXACT(LEFT("Keyboard", 3), "Key")` for a case-sensitive prefix. Explicitly choose case and culture policies; Unicode normalization, collation and grapheme segmentation are separate issues. In M, an explicit comparer such as `Comparer.Ordinal` or `Comparer.OrdinalIgnoreCase` makes the policy visible.

4.7 Dates and intervals

Add seven days to 2026-01-10; result 2026-01-17.

Power Fx

```
// Specify days explicitly; Date constructs a scalar
DateAdd(Date(2026, 1, 10), 7, TimeUnit.Days)
```

Power Query (M)

```
// A duration contains days, hours, minutes and seconds
#date(2026, 1, 10) + #duration(7, 0, 0, 0)
```

DAX

```
// Add days to a scalar date; not a date-table shift
DATE (2026, 1, 10) + 7
```

R

```r
# Date plus a numeric value adds days
as.Date("2026-01-10") + 7
```

Python

```python
# timedelta makes the unit of the offset explicit
date(2026, 1, 10) + timedelta(days=7)
```

PostgreSQL

```sql
-- An integer added to a date represents days
SELECT DATE '2026-01-10' + 7;
```

Date arithmetic is distinct from shifting a DAX date filter with `DATEADD()`. The latter returns a set of dates, not a scalar timestamp. Time zones, daylight saving and calendar-month arithmetic require separate tests.

4.8 Type conversion

Convert the string `"42"` into a number.

Power Fx

```
// Value interprets text according to parsing settings
Value("42")
```

Power Query (M)

```
// Pass the culture explicitly to the parser
Number.FromText("42", "en-US")
```

DAX

```
// VALUE converts a text scalar in model context
VALUE ("42")
```

R

```r
# Convert to a numeric vector; invalid text can warn
as.numeric("42")
```

Python

```python
# int expects integer text, not decimal text
int("42")
```

PostgreSQL

```sql
-- The double colon is PostgreSQL cast syntax
SELECT '42'::integer;
```

Locale-sensitive parsing must be controlled at ingestion. In PostgreSQL an invalid cast fails; R may warn and return `NA`. Python `int()` rejects fractional strings. Explicit schema validation is preferable to silently coercing invalid values.

4.9 Missing values

Substitute zero for a missing numeric scalar.

Power Fx

```
// Coalesce skips blank values
Coalesce(Blank(), 0)
```

Power Query (M)

```
// The null-coalescing operator supplies the fallback
null ?? 0
```

DAX

```
// COALESCE returns the first nonblank expression
COALESCE (BLANK (), 0)
```

R

```
# dplyr coalesce handles typed missing values
coalesce(NA_real_, 0)
```

Python

```
# Test identity with None, not truthiness
x = None
# For nullable pandas data, fill only the missing entries
values = pd.Series([None, 0, 5], dtype="Float64")
filled = values.fillna(0)
# The scalar equivalent returns zero for None
0 if x is None else x
```

PostgreSQL

```
-- Cast NULL to numeric before selecting the fallback
SELECT COALESCE(NULL::numeric, 0);
```

The Python example assumes `x = None`. For pandas, use `fillna()`; in the example, `filled` contains `0`, `0`, and `5`. Never use `x or 0` if falsey values are meaningful. Zero and missing data are different business states; use this replacement only when justified.

4.10 Error handling

Try to convert invalid text to a number and return a missing value.

Power Fx

```
// IfError supplies a fallback for a conversion error
IfError(Value("oops"), Blank())
```

Power Query (M)

```
// try/otherwise catches the conversion error
try Number.FromText("oops") otherwise null
```

DAX

```
// IFERROR handles a scalar expression error
IFERROR (VALUE ("oops"), BLANK ())
```

R

```r
# Suppress the conversion warning; result is NA
suppressWarnings(as.numeric("oops"))
```

Python

```python
# Catch the specific exception; do not hide unrelated errors
try:
    x = float("oops")
# Handle only the expected conversion failure
except ValueError:
    x = None
```

PostgreSQL

```sql
-- Validate integer text before attempting the cast
SELECT CASE WHEN s ~ '^[0-9]+$' THEN s::numeric ELSE NULL END
FROM (VALUES ('oops')) AS v(s);
```

The SQL predicate is only an integer validation pattern, not a general numeric parser. For production, stage invalid records with reason codes. Power Fx error handling depends on formula-level error management; errors should be visible to users when writes fail.

4.11 Filter rows

Keep orders with `Revenue >= 300`; IDs 4, 5, 7, 10.

Power Fx

```
// Filter evaluates the predicate in each record scope
Filter(Sales, Revenue >= 300)
```

Power Query (M)

```
// each binds the current row to the implicit parameter
Table.SelectRows(Sales, each [Revenue] >= 300)
```

DAX

```
// FILTER returns rows visible in the current context
FILTER (Sales, Sales[Revenue] >= 300)
```

R

```r
# dplyr resolves Revenue inside the data mask
Sales |>
  filter(Revenue >= 300)
```

Python

```python
# query evaluates the expression against frame columns
Sales.query("Revenue >= 300")
```

PostgreSQL

```sql
-- WHERE restricts rows before projection
SELECT * FROM Sales WHERE Revenue >= 300;
```

DAX `FILTER()` returns a table in existing context, not a modified measure context by itself. Power Fx `Filter()` against a remote connector must be delegable. SQL row order is not guaranteed without `ORDER BY`. pandas filtering preserves existing row order, but explicitly sort when the intended result requires a particular order.

4.12 Projection

Return `OrderID` and `Revenue` for all ten orders.

Power Fx

```
// ShowColumns returns a narrower table, not a mutation
ShowColumns(Sales, OrderID, Revenue)
```

Power Query (M)

```
// Column names are a list of text values
Table.SelectColumns(Sales, {"OrderID", "Revenue"})
```

DAX

```
// SELECTCOLUMNS projects a virtual table
SELECTCOLUMNS (Sales, "OrderID", Sales[OrderID], "Revenue", Sales[Revenue])
```

R

```
# select uses tidy-selection syntax
Sales |>
  select(OrderID, Revenue)
```

Python

```
# A list of names selects multiple columns
Sales[["OrderID", "Revenue"]]
```

PostgreSQL

```
-- SELECT projects columns without deduplicating
SELECT OrderID, Revenue FROM Sales;
```

Projection reduces width; it does not deduplicate rows. `SELECTCOLUMNS()` creates a virtual table; it does not alter the stored model. SQL projection can be pushed down to the source.

4.13 Create a column

Calculate 10% of `Revenue` as `Bonus`; order 1 yields 20.

Power Fx

```
// AddColumns evaluates Bonus for each record
AddColumns(Sales, Bonus, Revenue * 0.1)
```

Power Query (M)

```
// Supply a row function and a declared output type
Table.AddColumn(Sales, "Bonus", each [Revenue] * 0.1, type number)
```

DAX

```
// ADDCOLUMNS creates row context for the expression
ADDCOLUMNS (Sales, "Bonus", Sales[Revenue] * 0.1)
```

R

```r
# mutate adds a vector aligned to the original rows
Sales |>
  mutate(Bonus = Revenue * 0.1)
```

Python

```python
# assign returns a new frame
Sales.assign(Bonus=lambda d: d["Revenue"] * 0.1)
```

PostgreSQL

```sql
-- AS names a computed expression in the result
SELECT *, Revenue * 0.1 AS Bonus FROM Sales;
```

This is a row-level transformation. A DAX measure returning `SUM()` of the bonus is different from storing `Bonus` per row. M is preferable for static data shaping if folding and refresh semantics are appropriate.

4.14 Sort and take top N

Return orders 7, 5 and 4, breaking ties by `OrderID` ascending.

Power Fx

```
// Sort by revenue and use the ID to resolve ties
FirstN(
    SortByColumns(
        Sales,
        "Revenue", SortOrder.Descending,
        "OrderID", SortOrder.Ascending
    ),
    3
)
```

Because `FirstN` is nondelegable against data sources, the impact is not limited to performance. Using it against large remote sources can produce incorrect results because only a subset of records may be retrieved locally.

Power Query (M)

```
// Pass a list of sort specifications, then take three rows
Table.FirstN(
    Table.Sort(
        Sales,
        {{"Revenue", Order.Descending}, {"OrderID", Order.Ascending}}
    ),
    3
)
```

DAX

```
// TOPN selects rows; it does not guarantee display order
TOPN (
    3,
    Sales,
    Sales[Revenue], DESC,
    Sales[OrderID], ASC
)
```

R

```r
# arrange first, then take the first three rows
Sales |>
  arrange(desc(Revenue), OrderID) |>
  # The secondary key resolves ties before limiting rows
  slice_head(n = 3)
```

Python

```python
# Match each column to its ascending flag
(
    Sales
    .sort_values(["Revenue", "OrderID"], ascending=[False, True])
    # Resolve ties before limiting the result
    .head(3)
)
```

PostgreSQL

```sql
-- The secondary key makes LIMIT deterministic here
SELECT * FROM Sales ORDER BY Revenue DESC, OrderID LIMIT 3;
```

DAX `TOPN()` can return extra rows at the boundary when sort keys tie and does not guarantee output order. The unique secondary key here resolves the tie. Power Fx `FirstN()` is nondelegable in the documented canvas-app environment; sorting and taking the first N locally does not guarantee the global top N of a large remote source.

4.15 Unique values

Return the three `ProductID` values.

Power Fx

```
// Distinct returns a one-column Value table
Distinct(Sales, ProductID)
```

Power Query (M)

```
// Project the key before removing duplicate rows
Table.Distinct(Table.SelectColumns(Sales, {"ProductID"}))
```

DAX

```
// DISTINCT returns a virtual column table
DISTINCT (Sales[ProductID])
```

R

```r
# Keep one row per selected key
Sales |>
  distinct(ProductID)
```

Python

```python
# Select the key before dropping duplicates
Sales[["ProductID"]].drop_duplicates()
```

PostgreSQL

```
-- DISTINCT applies to the projected key
SELECT DISTINCT ProductID FROM Sales;
```

Power Fx `Distinct()` returns a single-column table named `Value` in current canvas apps. SQL `DISTINCT` works across the selected columns. DAX `VALUES()` can include a relationship-generated blank member representing unmatched facts or referential-integrity violations. This behavior differs from `DISTINCT()` and is important when analyzing incomplete dimensional relationships.

4.16 Aggregate an expression

Sum `Quantity * UnitPrice`; expected 2950.

Power Fx

```
// Sum evaluates the multiplication in record scope
Sum(Sales, Quantity * UnitPrice)
```

Power Query (M)

```
// Add a row expression, extract its list, then sum
List.Sum(Table.AddColumn(Sales, "Amount", each [Quantity]*[UnitPrice])[Amount])
```

DAX

```
// SUMX iterates visible fact rows
SUMX (Sales, Sales[Quantity] * Sales[UnitPrice])
```

R

```
# Multiply aligned vectors, then reduce
sum(Sales$Quantity * Sales$UnitPrice)
```

Python

```
# Multiply two aligned Series, then aggregate
(Sales.Quantity * Sales.UnitPrice).sum()
```

PostgreSQL

```
-- Aggregate the expression directly in the database
SELECT sum(Quantity * UnitPrice) FROM Sales;
```

`SUMX()` and Power Fx `Sum()` evaluate a record expression. M builds a column then aggregates a list. SQL can aggregate the expression directly. DAX measures evaluate within current filter context; expected result assumes no filters.

Sum of numeric input	100, 200, missing	All missing or empty typed input
R `sum()`	NA	NA if missing; 0 if empty
pandas `Series.sum()`	300	0 by default; use min_count=1 for missing result
PostgreSQL `SUM()`	300	NULL
DAX `SUM()`	300	BLANK
M `List.Sum()`	300	null

Power Fx aggregation also requires an explicit blank/error policy and host testing. In R use `na.rm = TRUE` only if ignoring missing observations is intended. Preserve the distinction between absence and a legitimate zero; validate all-missing and empty cases separately.

4.17 Grouped aggregation

Sum revenue by `ProductID`: P1=1300, P2=450, P3=1200.

Power Fx

```
// GroupBy creates nested tables; sum within each group
DropColumns(
    AddColumns(
        GroupBy(Sales, ProductID, Rows),
        Amount,
        Sum(Rows, Revenue)
    ),
    Rows
)
```

Power Query (M)

```
// The aggregation receives the grouped rows
Table.Group(
    Sales,
    {"ProductID"},
    {{"Amount", each List.Sum([Revenue]), type number}}
)
```

DAX

```
// The measure inherits the product group filter
SUMMARIZECOLUMNS (Products[ProductID], "Amount", [Total Revenue])
```

R

```
# Group the frame before reducing revenue
Sales |>
  group_by(ProductID) |>
  # Reduce each group to one row
  summarise(Amount = sum(Revenue), .groups = "drop")
```

Python

```
# Named aggregation defines the output column
(
    Sales
    .groupby("ProductID", as_index=False)
    # Named aggregation sets the result column name
    .agg(Amount=("Revenue", "sum"))
)
```

PostgreSQL

```
-- GROUP BY defines the aggregation grain
SELECT ProductID, sum(Revenue) AS Amount FROM Sales GROUP BY ProductID;
```

SQL expresses grouping explicitly; DAX uses the relationship from `Products` to `Sales` when evaluating the measure for each product. Power Fx `GroupBy()` creates nested tables; aggregation follows. Empty groups and missing identifiers can behave differently.

4.18 Many-to-one left lookup

Add `ProductName` to each order; order 1 is Keyboard.

Power Fx

```
// Alias the outer record to distinguish matching keys
AddColumns(
    Sales As s,
    ProductName,
    // Retrieve the matching record or requested field
    LookUp(Products, ProductID = s.ProductID, ProductName)
)
```

Power Query (M)

```
// NestedJoin returns a nested table; expand the name
Table.ExpandTableColumn(
    Table.NestedJoin(
        Sales, {"ProductID"}, Products, {"ProductID"},
        "P", JoinKind.LeftOuter
    ),
    "P",
    {"ProductName"}
)
```

DAX

```
// RELATED follows the existing many-to-one relationship
ADDCOLUMNS (Sales, "ProductName", RELATED (Products[ProductName]))
```

R

```
# Project the dimension before the left join
Sales |>
  left_join(
    Products |> select(ProductID, ProductName),
    by = "ProductID"
  )
```

Python

```
# Validate the many-to-one cardinality explicitly
Sales.merge(
    Products[["ProductID", "ProductName"]],
    on="ProductID",
    how="left",
    # Reject duplicate keys on the dimension side
    validate="many_to_one",
)
```

PostgreSQL

```
-- USING matches the shared key and retains left rows
SELECT s.*, p.ProductName FROM Sales s LEFT JOIN Products p USING (ProductID);
```

Power Fx `LookUp()` returns only the first match, so it is not a general relational join. `RELATED()` needs row context and an appropriate regular relationship to the lookup table; it cannot traverse a limited relationship. Enforce unique dimension keys; SQL and pandas can multiply rows if duplicates exist.

4.19 Append tables

Append `Sales` to itself; expected 20 rows.

Power Fx

```
// Table combines records from both input tables
Table(Sales, Sales)
```

An alternative implementation uses `Ungroup()` to flatten explicitly nested tables. Although unnecessary for this simple append operation, this pattern is useful when working with nested table structures.

Power Fx

```
// Nest both tables as records, then flatten them into a single table
Ungroup(
    Table(
        {Rows: Sales},
        {Rows: Sales}
    ),
    Rows
)
```

Power Query (M)

```
// Combine tables using their column names
Table.Combine({Sales, Sales})
```

DAX

```
// UNION retains duplicates and aligns by position
UNION (Sales, Sales)
```

R

```
# Bind rows by column names
bind_rows(Sales, Sales)
```

Python

```
# Reset the concatenated frame index
pd.concat([Sales, Sales], ignore_index=True)
```

PostgreSQL

```
-- UNION ALL deliberately keeps duplicates
SELECT * FROM Sales UNION ALL SELECT * FROM Sales;
```

DAX `UNION()` retains duplicates and matches columns by position. SQL `UNION ALL` retains duplicates, while `UNION` eliminates them. M and pandas usually align columns by names; schema mismatches deserve an explicit check.

SQL and DAX tables can retain duplicate rows (bag semantics), while Python sets and R set operations discard multiplicity. If an intersection must retain counts, specify whether the count is that of the left input (DAX INTERSECT) or the minimum of both input counts (SQL INTERSECT ALL). The distinct-key drill intentionally avoids this difference.

4.20 Set intersection

Common product IDs between orders in 2025 and 2026; P1, P2, P3.

Power Fx

```
// Use distinct local IDs and an alias for membership
Filter(
    Distinct(Filter(Sales, Year(OrderDate)=2025), ProductID) As p,
    p.Value in Distinct(
        Filter(Sales, Year(OrderDate)=2026), ProductID
    ).Value
)
```

Power Query (M)

```
// Extract two distinct key lists and intersect them
List.Intersect({
    List.Distinct(
        Table.SelectRows(Sales, each Date.Year([OrderDate])=2025)[ProductID]
    ),
    List.Distinct(
        Table.SelectRows(Sales, each Date.Year([OrderDate])=2026)[ProductID]
    )
})
```

DAX

```
// CALCULATETABLE changes year context before VALUES
INTERSECT (
    CALCULATETABLE (VALUES (Sales[ProductID]), Calendar[Year] = 2025),
    CALCULATETABLE (VALUES (Sales[ProductID]), Calendar[Year] = 2026)
)
```

R

```
# Build two year-specific vectors and intersect them
intersect(Sales$ProductID[year(Sales$OrderDate) == 2025],
    Sales$ProductID[year(Sales$OrderDate) == 2026])
```

Python

```
# Convert both filtered Series into sets
(
    set(Sales.loc[Sales.OrderDate.dt.year == 2025, "ProductID"])
    & set(Sales.loc[Sales.OrderDate.dt.year == 2026, "ProductID"])
)
```

PostgreSQL

```
-- INTERSECT removes duplicate result keys
SELECT ProductID
FROM Sales
WHERE extract(year from OrderDate)=2025
INTERSECT SELECT ProductID
FROM Sales
WHERE extract(year from OrderDate)=2026;
```

DAX `INTERSECT()` retains duplicates from its first argument, so `VALUES()` is intentional. SQL `INTERSECT` deduplicates unless `ALL` is specified. The Power Fx expression assumes local data; nested record aliases prevent ambiguity.

4.21 Nested values

Read the `city` field from a nested record; result `"San Jose"`.

Power Fx

```
// With binds a record containing another record
With({r:{city:"San Jose"}}, r.city)
```

Power Query (M)

```
// Square brackets construct and access a record
let r = [city="San Jose"] in r[city]
```

DAX

No general nested record value; normalize into a table.

R

```
# A named list exposes the city element
list(city="San Jose")$city
```

Python

```
# Index a dictionary by its string key
{"city":"San Jose"}["city"]
```

PostgreSQL

```
-- Cast JSON text to jsonb; ->> extracts text
SELECT '{"city":"San Jose"}'::jsonb ->> 'city';
```

M records and PostgreSQL `jsonb` are not the same abstraction: M values can contain functions and tables, while JSON is a serialization format. DAX expects tabular data; expand nested structures upstream.

4.22 Reshape wide

Pivot `ProductID` rows into columns of revenue by year; P1/P2/P3 in 2025: 600/150/200; in 2026: 700/300/1000.

Power Fx

Use a gallery with per-product `Sum(Filter(...))` or explicit `AddColumns()`; no general pivot primitive.

Power Query (M)

```
// Project the pivot grain and aggregate duplicate cells
Table.Pivot(
    Table.SelectColumns(
        Table.AddColumn(Sales, "Year", each Date.Year([OrderDate])),
        {"Year", "ProductID", "Revenue"}
    ),
    {"P1", "P2", "P3"},
    "ProductID",
    "Revenue",
    List.Sum
)
```

DAX

Use matrix: `Calendar[Year]` rows, `Products[ProductID]` columns, `[Total Revenue]` values.

R

```r
# values_fn resolves multiple observations per cell
Sales |>
  mutate(Year = year(OrderDate)) |>
  select(Year, ProductID, Revenue) |>
  # Multiple orders in a cell must be aggregated
  pivot_wider(
    names_from = ProductID,
    values_from = Revenue,
    values_fn = sum
  )
```

Python

```python
# Derive year before pivoting and summing cells
(
    Sales
    .assign(Year=lambda d: d["OrderDate"].dt.year)
    # Aggregate multiple orders within each year/product cell
    .pivot_table(
        index="Year", columns="ProductID",
        values="Revenue", aggfunc="sum"
    )
)
```

PostgreSQL

```sql
-- FILTER restricts each aggregate to its product
SELECT
    EXTRACT(YEAR FROM OrderDate) AS yr,
    -- Each filtered aggregate defines one output column
    SUM(Revenue) FILTER (WHERE ProductID = 'P1') AS p1,
    SUM(Revenue) FILTER (WHERE ProductID = 'P2') AS p2,
    SUM(Revenue) FILTER (WHERE ProductID = 'P3') AS p3
FROM Sales
-- Set the grain of the aggregate result
GROUP BY yr;
```

The DAX example uses a visual layout, not a materialized wide table. M unpivot and tidyr `pivot_longer()` reverse a wide schema into rows. PostgreSQL conditional aggregation requires known output columns; crosstab is an optional extension.

4.23 Map and reduce

Square 1 through 5 and sum; expected 55.

Power Fx

```
// Sequence produces a Value column; Sum reduces it
Sum(Sequence(5), Value^2)
```

Power Query (M)

```
// each maps the implicit parameter before reduction
List.Sum(List.Transform({1..5}, each Number.Power(_, 2)))
```

DAX

```
// GENERATESERIES supplies the iterator table
SUMX (GENERATESERIES (1, 5), [Value]^2)
```

R

```
# map_dbl returns a numeric vector for sum
sum(map_dbl(1:5, function(x) x^2))
```

Python

```
# A scalar alternative is sum(x**2 for x in range(1, 6))
# Vectorize the square over a NumPy array
np.square(np.arange(1, 6)).sum()
```

PostgreSQL

```
-- generate_series supplies rows to the aggregate
SELECT sum(n*n) FROM generate_series(1, 5) AS g(n);
```

Mapping returns a transformed sequence; reduction collapses it to one value. Power Fx `ForAll` may process records in an unspecified order, so do not rely on sequencing side effects. DAX iterators are expressions, not mutable loops.

4.24 Reusable function

Define `square()` and evaluate it at 5; result 25.

Power Fx

```
// Declare parameter and return types in App.Formulas. Invoke Square(5) separately
Square(x: Number): Number = x^2;
```

Power Query (M)

```
// A typed anonymous function can be bound or invoked
let
    Square = (x as number) as number => Number.Power(x, 2)
in
    Square(5)
```

DAX

As of the June 2026 release, DAX UDFs are generally available in Power BI Desktop and Power BI Service and require compatibility level 1702 or higher [24]. Define a query-local function and invoke it in a result table:

DAX

```
DEFINE
    FUNCTION Square = (x : SCALAR) => x * x
EVALUATE
    ROW ("Result", Square (5))
```

Result: 25. A persisted model function is defined in the semantic model rather than only in a query. Check the target host and compatibility level before using this feature.

R

```
# Define a function with a single formal parameter
square <- function(x) x^2
square(5)
```

Python

```python
# Define a function and return the expression
def square(x):
    """Return the square of a numeric value."""
    return x**2


# Evaluate the function on a scalar
square(5)
```

PostgreSQL

```sql
-- IMMUTABLE declares a contract for the optimizer
CREATE FUNCTION square(x numeric)
RETURNS numeric LANGUAGE sql IMMUTABLE
AS $$ SELECT x*x $$;
SELECT square(5);
```

M functions can be passed to other functions. PostgreSQL `IMMUTABLE` is a promise to its optimizer, not a guarantee enforced by the compiler. Power Fx and DAX UDF syntax and availability should be checked in the actual target host.

4.25 Ranking

Dense rank products by revenue; P1=1, P3=2, P2=3.

Power Fx

```
// Rank each grouped total by counting distinct greater totals
With(
    {Totals: DropColumns(
        AddColumns(GroupBy(Sales, ProductID, Rows),
            Amount, Sum(Rows, Revenue)), Rows)},
    AddColumns(
        Totals As p,
        Rank,
        1 + CountRows(Distinct(
            Filter(Totals As q, q.Amount > p.Amount), Amount))
    )
)
```

Power Query (M)

```
let
    Grouped = Table.Group(Sales, {"ProductID"},
        {{"Amount", each List.Sum([Revenue]), type number}}),
    // Sort distinct totals so equal totals receive equal ranks
    Amounts = List.Sort(List.Distinct(Grouped[Amount]), Order.Descending),
    Ranked = Table.AddColumn(Grouped, "Rank",
        each List.PositionOf(Amounts, [Amount]) + 1, Int64.Type)
in
    Ranked
```

DAX

```
// ALLSELECTED defines the comparison universe
IF (ISINSCOPE (Products[ProductID]),
    RANKX (ALLSELECTED (Products[ProductID]),
        [Total Revenue], , DESC, DENSE))  /* measure */
```

R

```
# dense_rank assigns consecutive ranks to distinct totals
Sales |>
  group_by(ProductID) |>
  summarise(Amount = sum(Revenue), .groups = "drop") |>
  # Rank the totals rather than the original orders
  mutate(Rank = dense_rank(desc(Amount)))
```

Python

```
# Aggregate before ranking; ties share dense ranks
(
    Sales
    .groupby("ProductID")["Revenue"]
    .sum()
    # Rank grouped totals; ties share a rank
    .rank(method="dense", ascending=False)
)
```

PostgreSQL

```
-- Rank the grouped results using a window expression
SELECT ProductID, dense_rank() OVER (ORDER BY sum(Revenue) DESC) AS rank
FROM Sales
-- Set the grain of the aggregate result
GROUP BY ProductID;
```

Ranking and sorting are not identical. The Power Fx and M examples assign dense ranks explicitly; repeated scans or list searches can be costly at large cardinalities. The DAX measure assumes a visual grouped by ProductID, not ProductName; use the same grouping column in ISINSCOPE and ALLSELECTED (see the ProductName variant in Chapter 7). `RANKX()` itself introduces row context while iterating that universe.

4.26 Descriptive statistics

Median order revenue; expected 200.

Power Fx

```
// Median evaluates the record expression
Median(Sales, Revenue)
```

Power Query (M)

```
// Extract the Revenue list before computing the median
List.Median(Sales[Revenue])
```

DAX

```
// MEDIAN operates within the filter context
MEDIAN (Sales[Revenue])
```

R

```r
# Compute the median of the revenue vector
median(Sales$Revenue)
```

Python

```python
# Compute the median of a Series
Sales.Revenue.median()
```

PostgreSQL

```sql
-- percentile_cont interpolates at the median
SELECT percentile_cont(0.5) WITHIN GROUP (ORDER BY Revenue) FROM Sales;
```

Median and percentile interpolation rules should be checked for missing values and even sample sizes. A DAX measure computes over the current filter context. For inference, uncertainty and regression diagnostics, R or Python remains more suitable.

4.27 API ingestion

Fetch a JSON response and parse it; no live API is called by this drill.

Power Fx

Use a configured connector or flow; `ParseJSON()` on a returned text value.

Power Query (M)

```
// Evaluate the expression in the indicated host
Json.Document(Web.Contents("https://api.example.com/items"))
```

DAX

No native HTTP ingestion; use M or an upstream service.

R

```r
# jsonlite handles retrieval and JSON parsing
jsonlite::fromJSON("https://api.example.com/items")
```

Python

```python
# The URL is illustrative; set a request timeout. Import requests before use
url = "https://api.example.com/items"
response = requests.get(url, timeout=10)
response.raise_for_status()
response.json()
```

PostgreSQL

Core SQL has no general HTTP client; use an ETL service or an approved extension.

The URL is illustrative. Production API code needs authentication, timeouts, status checks, retries, pagination and schema validation. Keep secrets out of queries, code and exported artifacts.

4.28 State and durable writes

Update order 1 quantity to 3; illustrative, do not run on the fixture if following later drills.

Power Fx

```
// Patch targets the record matching the order key
Patch(Sales, LookUp(Sales, OrderID = 1), {Quantity:3})
```

Power Query (M)

Return a transformed table; M is not a general transactional write language.

DAX

No transactional mutation from a measure or query.

R

```
# Rebind the local frame; no database write occurs
Sales <- Sales |>
  mutate(Quantity = if_else(OrderID == 1, 3L, Quantity))
```

Python

```
# Assign only the matching rows in the local frame
Sales.loc[Sales.OrderID == 1, "Quantity"] = 3
```

PostgreSQL

```
-- UPDATE changes transactional rows; commit determines persistence
UPDATE Sales SET Quantity = 3 WHERE OrderID = 1;
```

Power Fx `Patch()` against a collection changes local state, while against a connected data source it attempts a durable write. R and pandas examples alter local data only. PostgreSQL provides transactions, constraints and concurrency semantics. The setup uses a generated `Revenue` column in PostgreSQL; other environments must recompute the derived value after changing `Quantity`.

4.29 Compose a transformation pipeline

Select orders with quantity at least 3, calculate a new `Amount` equal to `Quantity` times `UnitPrice`, keep `OrderID` and `Amount`, and sort by `OrderID`. Expected IDs: 2, 4, 5, 7, 8; amounts: 150, 400, 500, 600, 200.

Power Fx

```
// Compose filter, extension, projection and sorting
Sort(
    ShowColumns(
        AddColumns(
            Filter(Sales, Quantity >= 3),
            Amount,
            Quantity * UnitPrice
        ),
        OrderID, Amount
    ),
    OrderID
)
```

Power Query (M)

```
// Name each immutable intermediate table
let
    // Select rows before deriving amounts
    F = Table.SelectRows(Sales, each [Quantity] >= 3),
    // The row lambda computes a value for every retained row
    A = Table.AddColumn(F, "Amount",
        each [Quantity] * [UnitPrice]),
    // Drop columns that are not part of the result contract
    P = Table.SelectColumns(A, {"OrderID", "Amount"})
in
    Table.Sort(P, {{"OrderID", Order.Ascending}})
```

DAX

```
// Project the filtered virtual table; sort in Query View
SELECTCOLUMNS (FILTER (Sales, Sales[Quantity] >= 3),
    "OrderID", Sales[OrderID], "Amount", Sales[Quantity] * Sales[UnitPrice])
```

R

```
# Compose transformations in reading order
Sales |>
  filter(Quantity >= 3) |>
  # Calculate the amount only for retained records
  mutate(Amount = Quantity * UnitPrice) |>
  select(OrderID, Amount) |>
  arrange(OrderID)
```

Python

```
# Chain transformations without mutating the input
(
    Sales
    .query("Quantity >= 3")
    # Derive a column from the current intermediate frame
    .assign(Amount=lambda d: d["Quantity"] * d["UnitPrice"])
    .loc[:, ["OrderID", "Amount"]]
    .sort_values("OrderID")
)
```

PostgreSQL

```
-- Select the computed expression and order the result
SELECT OrderID, Quantity * UnitPrice AS Amount
FROM Sales
WHERE Quantity >= 3
ORDER BY OrderID;
```

For DAX, put `EVALUATE` before the table expression and append `ORDER BY [OrderID] ASC` in Query View. M's named steps are useful for inspection and editing; Power Fx nested expressions can be refactored with `With()`. In R, the native pipe offers readable left-to-right composition, while the equivalent pandas expression chains methods. SQL states the desired result, leaving the physical operator order to the planner. Do not infer actual execution cost from the visual order of steps.

4.30 Running revenue

Compute revenue cumulatively by ascending `OrderDate`, breaking any same-day ties with `OrderID`. The first three cumulative amounts are 200, 350, and 550; the last is 2,950.

Power Fx

```
// Compare the inner row to the outer row and sum
AddColumns(
    Sales As s,
    Running,
    Sum(
        Filter(
            Sales As t,
            t.OrderDate < s.OrderDate Or
            (t.OrderDate = s.OrderDate And t.OrderID <= s.OrderID)
        ),
        Revenue
    )
)
```

Power Query (M)

Sort, then accumulate a revenue list with `List.Accumulate()`; for large inputs prefer a folded native query or more efficient implementation.

DAX

```
VAR d =
// Capture the current row keys before scanning all rows
Sales[OrderDate]
VAR id = Sales[OrderID]
// Evaluate the final expression using the bound variables
RETURN
    // Iterate all earlier rows, including the current order
    SUMX (
        FILTER (ALL (Sales),
            OR (Sales[OrderDate] < d,
                AND (Sales[OrderDate] = d, Sales[OrderID] <= id))),
        Sales[Revenue]
    )
```

R

```
# Sort by both keys before calculating cumulative sums
Sales |>
  arrange(OrderDate, OrderID) |>
  # Accumulate in the established chronological order
  mutate(Running = cumsum(Revenue))
```

Python

```
# Sort first so cumsum uses the intended row order
(
    Sales
    .sort_values(["OrderDate", "OrderID"])
    # Cumulative sums follow the sorted row order
    .assign(Running=lambda d: d["Revenue"].cumsum())
)
```

PostgreSQL

```
-- ROWS defines a physical cumulative window frame
SELECT *, sum(Revenue) OVER (ORDER BY OrderDate, OrderID
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS Running
FROM Sales
ORDER BY OrderDate, OrderID;
```

The DAX expression is a calculated column illustrating variables captured in an outer row context, not an interactive running-total measure. In a production semantic model, a date-aware measure using `CALCULATE()` is preferable when the cumulative result must respond to selected periods. Power Fx's repeated scan and a similar naive M algorithm can scale quadratically. PostgreSQL's window operation and vectorized R/Python cumulative sums are better starting points for a fixed running-total transformation. Explicit `ROWS` framing avoids surprises with peer rows.

A repeated full scan for each of n rows is $O(n^2)$ in a straightforward implementation. Sorting followed by a linear cumulative pass has $O(n \log n)$ comparison-sort cost and $O(n)$ accumulation cost; already ordered data can avoid the sort. These are algorithmic bounds, not measured runtimes or guarantees about a particular optimizer. The outer SQL ORDER BY specifies presentation order independently of the window frame.

4.31 Reshape long

Transform the two numeric fields `Quantity` and `UnitPrice` into rows with the fields `OrderID`, `Metric` and `Value`. Order 1 becomes two rows: `Quantity = 2` and `UnitPrice = 100`. The fixture produces 20 rows.

Power Fx

```
// Build one table per metric, then append
Table(
    ShowColumns(
        AddColumns(Sales, Metric, "Quantity", Value, Quantity),
        OrderID, Metric, Value
    ),
    ShowColumns(
        AddColumns(Sales, Metric, "UnitPrice", Value, UnitPrice),
        OrderID, Metric, Value
    )
)
```

Power Query (M)

```
// Project identifiers and unpivot the two numeric fields
Table.Unpivot(
    Table.SelectColumns(Sales, {"OrderID", "Quantity", "UnitPrice"}),
    {"Quantity", "UnitPrice"},
    "Metric",
    "Value"
)
```

DAX

```
// UNION appends two projected virtual tables
UNION (
    SELECTCOLUMNS (
        Sales,
        "OrderID", Sales[OrderID],
        "Metric", "Quantity",
        "Value", Sales[Quantity]
    ),
    SELECTCOLUMNS (
        Sales,
        "OrderID", Sales[OrderID],
        "Metric", "UnitPrice",
        "Value", Sales[UnitPrice]
    )
)
```

R

```
# Pivot the selected measure columns into rows
Sales |>
  select(OrderID, Quantity, UnitPrice) |>
  # Turn the two numeric fields into metric/value rows
  pivot_longer(
    cols = c(Quantity, UnitPrice),
    names_to = "Metric",
    values_to = "Value"
  )
```

Python

```
# melt retains the identifier and emits metric/value pairs
Sales.melt(
    id_vars="OrderID",
    value_vars=["Quantity", "UnitPrice"],
    # Name the two columns created by unpivoting
    var_name="Metric",
    value_name="Value",
)
```

PostgreSQL

```
-- LATERAL expands two values for each source row
SELECT s.OrderID, v.Metric, v.Value
FROM Sales s
CROSS JOIN LATERAL (
    VALUES ('Quantity', s.Quantity), ('UnitPrice', s.UnitPrice)
) AS v(Metric, Value);
```

Unpivot changes the grain. It can be convenient for uniform display but is not a reason to mix incompatible units in a single aggregate. In M, null-valued cells may be omitted by unpivot; SQL `VALUES`, pandas `melt()` and tidyr `pivot_longer()` may retain missing observations unless explicitly filtered. This example has no missing values, so the expected shape is identical. For heterogeneous column types, choose an explicit common representation and avoid accidental coercion.

Takeaway. Similar syntax is not equivalent execution. Verify input types, output shape, duplicate handling, missing values, filter context, and whether the operation reads or writes persistent data.

5 Power Fx: application behavior and reactive formulas

5.1 The Excel analogy and its limits

A formula attached to a property describes what that property should be. For example, set a gallery's `Items` property to `Filter(Sales, Quantity >= Slider1.Value)` and a label's `Text` property to `Sum(Gallery1.AllItems, Revenue)`. The gallery automatically responds to the slider, much like a spreadsheet dependency or Shiny reactive expression. The label example sums only loaded gallery items, not necessarily the complete remote result; for a real KPI calculate against an appropriately delegable source instead.

Construct	Purpose	Familiar comparison
Property formula	Derive UI state	Shiny reactive expression
`App.Formulas`	Named derived values	Centralized reactive definitions
`OnSelect`	Trigger behavior	Shiny `observeEvent()`; Python event handler
`Set()`	App-wide variable	Mutable application state
`UpdateContext`	Screen-local context	Scoped view state
`With()`	Local immutable bindings	M `let`; R local expression
`Collect` / `ClearCollect`	Modify local collection	In-memory table mutation
`Patch()`	Modify records	API write or SQL `UPDATE` (not equivalent guarantees)

The key design principle is to minimize independently mutable state. A variable holding a calculated total can become stale when inputs change, whereas a property formula or named formula can maintain dependencies. Collections are useful for a draft or bounded cache but do not automatically provide database transactions, isolation or distributed synchronization. Power Fx is open source as a language project, but the available functions and connectors differ across host environments [1,2].

Use named formulas for derived values and behavior formulas for event-triggered effects. A named formula describes a dependency rather than a sequence of assignments; a Set variable is explicitly updated and can therefore drift from its inputs. Keep writes out of derived display calculations.

5.1.1 Following a reactive dependency in practice

Imagine a slider initially set to 2. A gallery's `Items` formula references the slider; a label references the gallery. Changing the slider to 3 changes an input to the gallery formula, and the runtime updates the dependent results. No explicit loop is needed. The dependency graph is the set of these relationships between properties, named formulas and other inputs. A button that writes a record is different: its `OnSelect` behavior runs because an event occurs, not because every upstream formula is recalculated.

If you instead calculate a total once using `Set()`, the variable contains the value computed at that moment. Changing a slider does not automatically repeat that assignment. A named formula or property expression is usually preferable for derived state. Use a variable for state that must persist independently of its inputs, such as the selected quiz question or a draft submission. `UpdateContext()` narrows the state to a screen; `With()` introduces temporary names for a single expression.

A connector is an adapter between the application and another system. It exposes operations, schemas, authentication and sometimes query translation. It does not erase differences between a relational database, a list service and an

API. A `Patch()` operation may fail because of permission, validation, concurrency or network conditions. An app should handle these failures and the underlying service should enforce durable integrity. A successful local validation is not proof that a remote write succeeded.

Record scope means that a function evaluates its argument with a particular record available. For a table of orders, `Filter()` visits rows and can resolve `Quantity` from each row. A nested `LookUp()` introduces another scope. Use `As` aliases to state which record supplies a field; otherwise matching identical column names can accidentally compare an inner field with itself.

5.2 Record scopes, nested expressions and validation

The following formula builds a display table using a scoped alias. A nested `LookUp()` otherwise risks comparing the dimension's `ProductID` to itself rather than to the outer `Sales` row.

Power Fx

```
// Alias the outer order and inner product to avoid ambiguous keys
AddColumns(
    Sales As order,
    DisplayName,
    // Retrieve the matching record or requested field
    LookUp(Products As product,
        product.ProductID = order.ProductID,
        product.ProductName)
)
```

The key distinction from SQL aliasing is that `LookUp()` is an expression returning a single match, not a join returning every matching pair. A missing match returns blank. Scope disambiguation is especially important in nested `Filter()`, `ForAll`, `AddColumns()` and `LookUp()` expressions. A meaningful alias often prevents subtle errors that are hard to diagnose in a low-code editor.

For a quiz screen, put this formula in a classic text input's associated submit button `DisplayMode` property (modern controls may expose different value properties):

Power Fx

```
// Disable submission until the trimmed answer is nonblank
If(IsBlank(TrimEnds(AnswerInput.Text)),
    DisplayMode.Disabled,
    DisplayMode.Edit)
```

Then use a separate `OnSelect` behavior formula. Assume a configured `Attempts` source with the fields below, a current word record and a valid answer. The fixture sales tables are not involved.

Power Fx

```
// Try a connector write; report failure or success to the user
IfError(
    // Defaults creates the base record for a new attempt
    Patch(Attempts, Defaults(Attempts), {
        WordID: CurrentWord.WordID,
        Answer: AnswerInput.Text,
        Correct: Lower(TrimEnds(AnswerInput.Text)) =
                 Lower(TrimEnds(CurrentWord.ExpectedAnswer))
    }),
    Notify("Could not save the attempt", NotificationType.Error); false,
    Notify("Saved", NotificationType.Success); true
)
```

UI validation is a convenience, not a security boundary. The server must enforce permissions, schema constraints, uniqueness, and any integrity rules. The example normalizes whitespace and case but does not handle accents,

alternative valid translations or linguistic ambiguity. A real learning app needs explicit normalization and feedback rules. Avoid marking AI-generated evaluation as definitive without review.

The explicit final IfError argument runs on success; both returned branches are Boolean, so the Patch record is not returned. Formula-level error management must be enabled [25]. A failed or timed-out connector response does not prove that a write was never committed. Use a stable submission key, storage-side uniqueness, and a reconciliation strategy for ambiguous outcomes; refresh stale data and use source-supported concurrency controls where available.

For clarity, here are the two reactive property formulas discussed above.

Gallery Items

```
// Re-evaluate when the slider changes
Filter(Sales, Quantity >= Slider1.Value)
```

Label Text

```
// Only loaded gallery items contribute to this total
Sum(Gallery1.AllItems, Revenue)
```

5.3 Delegation lab

Suppose there are 100,000 attempts, and a user filters all records from the last week. A connector may delegate a simple date predicate and return matching pages efficiently. Wrapping the date in a custom transformation, collecting the full source first, or using an unsupported expression may break delegation. The resulting partial dataset may be indistinguishable from a genuinely complete set on a small test screen.

Test	What to observe
Local fixture with ten records	Formula correctness and UI behavior
Source with more than local limit	Missing records or delegation warnings
Indexed predicate on source	Remote query latency and correctness
Failed write, revoked permission	Error feedback and lack of false success
Duplicate submission	Idempotency, uniqueness and race conditions

This testing approach resembles contract and integration testing more than ordinary unit testing. Test formula behavior independently where possible, then validate the connector, data size, authorization and app environment. Availability of Test Studio, test automation, source export and UDFs is host-specific; confirm the target Power Platform environment rather than assuming features from a demo apply everywhere.

Takeaway. Power Fx adds a managed reactive UI and connector-aware business logic. Its main learning challenge is not arithmetic or filtering, but reliable state, record scopes, delegation, and application lifecycle.

5.3.1 Diagnosing a partial result

For a concrete test, set the nondelegable row limit to a low value in a development app, then construct a query whose only matching record is beyond the records read locally. Compare the result with a direct query against the source. The local formula may be perfectly correct over the records it received and still be wrong for the entire dataset. Increasing the limit is not a general solution: eventually the source grows past it. Prefer supported delegable predicates, a server-side view or an appropriately bounded query. Delegation support is specific to the connector, operation and data type; inspect the editor's warnings and test actual results [2].

6 Power Query (M): a functional integration language

6.1 A complete pipeline

Power Query (M)

```
// Assign source types before filtering and computing the amount
let
    // Apply the declared column types
    Typed = Table.TransformColumnTypes(Sales, {
        {"OrderID", Int64.Type}, {"OrderDate", type date},
        {"Quantity", Int64.Type}, {"UnitPrice", type number}
    }),
    // each evaluates the predicate for one row
    Filtered = Table.SelectRows(Typed, each [Quantity] >= 3),
    // Extend the filtered table; earlier bindings are unchanged
    Added = Table.AddColumn(Filtered, "Amount",
        each [Quantity] * [UnitPrice], type number),
    // Project the output columns, then establish order
    Result = Table.Sort(
        Table.SelectColumns(Added, {"OrderID", "Amount"}),
        {{"OrderID", Order.Ascending}})
in
    Result
```

Expected order IDs: 2, 4, 5, 7, 8; amounts: 150, 400, 500, 600, 200. M does not mutate Typed when Filtered is defined. The Applied Steps interface presents these bindings as sequential steps, but the language evaluates dependencies. A reference to `Sales` in this query must refer to a separate source query, not the name of the current query itself.

Technique	M expression	R/Python/PostgreSQL analogy
Local named step	`let A = ... in A`	R binding; pandas chain; SQL CTE
Lambda	`(x) => x * 2`	R function; Python lambda
Shorthand	`each [Quantity] * 2`	Anonymous function on a row
Type declaration	`type number`	Schema or type annotation (different enforcement)
Record access	`[Quantity]` in row scope	Named field / dict access
Positional access	`{0}`	Python index 0
Accumulate	`List.Accumulate()`	purrr reduce; functools.reduce

`each` is shorthand for a one-argument function whose parameter is `_`. In a table row scope, `each [Quantity]` is equivalent to `(_) => _[Quantity]`. Explicit parameters are clearer in nested transformations. M permits recursive functions with inclusive references, written with `@`, but recursion is rarely the best way to express a table transformation.

6.1.1 Reading the M pipeline as a dependency graph

The result binding depends on `Added`; `Added` depends on `Filtered`; `Filtered` depends on `Typed`. The engine can trace those dependencies backwards from `in Result`. Names are lexical bindings, not mutable intermediate variables. Referencing an earlier binding does not overwrite its value. Binding names must be unique within a binding

environment; nested environments can shadow outer names. Nor is there a guarantee that the engine materializes every named table: streaming, source translation and internal buffering can change the physical execution strategy.

A record has named fields, while a list is an ordered sequence of values. A table has named columns and rows. A nested JSON object can become a record containing a list of records, which can then be expanded into rows. A binary value represents bytes, such as the content of a file or HTTP response; it is not automatically a table. `Json.Document()` interprets the payload, whereas `Table.FromRecords()` constructs a tabular result after parsing. These distinctions explain why a seemingly simple source import can involve several conversion steps.

An error in M is an evaluation outcome, not just a missing value. `try` allows you to inspect whether a calculation raised an error and recover or quarantine the affected input. Treat a genuine missing value differently from an invalid number or an inaccessible service. Replacing errors indiscriminately with null hides failures that should be diagnosed. A refresh pipeline should preserve enough information to identify rejected records and the step that rejected them.

A try around record construction may succeed because the field is evaluated later. Put error handling around the access that forces evaluation [26]:

Power Query (M)

```
let
    RecordValue = [Value = error "Invalid value"],
    Result = try RecordValue[Value] otherwise null
in
    Result
```

The result is null. A try around RecordValue alone does not necessarily catch that deferred field error.

6.2 Ingestion, nested values and reusable transforms

The following example parses a local literal JSON payload. It can be executed without a remote API or credentials.

Power Query (M)

```
// Parse literal JSON, expand the items list and assign types
let
    Payload = Json.Document("{""items"":[{""id"":1,""score"":9}]}"),
    // Extract the list of records and convert it to a table
    Items = Table.FromRecords(Payload[items]),
    // Apply the declared column types
    Typed = Table.TransformColumnTypes(Items,
        {{"id", Int64.Type}, {"score", Int64.Type}})
in
    Typed
```

Result: one row with id 1 and score 9. JSON text escapes quotes by doubling them in M. A real API response often contains nested records, lists, pagination tokens, and inconsistent schemas. Expand only required columns, validate the schema and preserve rejected records for diagnostics.

A reusable query named `NormalizeText` can contain:

Power Query (M)

```
// Accept nullable text; preserve null rather than converting it
(x as nullable text) as nullable text =>
    if x = null then null else Text.Lower(Text.Trim(x))
```

Apply the function to `Products`:

Power Query (M)

```
// Apply NormalizeText to the selected column and declare its type
Table.TransformColumns(
    Products,
    {{"ProductName", NormalizeText, type nullable text}}
)
```

Unlike Python mutation of a dataframe in place, the function returns a new table expression. The custom function may affect folding; check the actual connector and translation plan rather than assuming it is always folded or always local.

6.3 ETL design and source pushdown

Design choice	Recommendation	Why
Relational joins on large PostgreSQL tables	Use database views or folding	Reuse optimizer, indexes, constraints
Combine departmental Excel files	Use M folder connector and typed transform	Repeatable self-service ingestion
Schema drift in external API	Stage and validate	Avoid quietly dropping or coercing records
Cross-source privacy	Classify sources; respect privacy firewall	Prevent unintended information transfer
`Table.Buffer()`	Use only with measured reason	Can block folding, increase memory
Data refresh	Schedule and monitor in host	M expression alone is not orchestration

For example, a PostgreSQL view that normalizes shared operational records can be reused by Power BI, Python services and R analysts. M remains useful for combining those records with an approved spreadsheet or API. When database credentials permit, prefer a read-only view or parameterized native query; avoid SQL strings built from unsanitized user inputs. Query folding is an optimization, not a permission system. Data privacy settings, gateways and connector capabilities are separate from M syntax.

The Data Privacy Firewall partitions a query and regulates how partitions reference one another and access sources, to prevent unintended transfer of data between sources during folding. In the standard configuration, a partition that references another partition cannot also directly access a source; compatible privacy levels and host-specific settings affect this behavior. Diagnose the partition and source privacy settings instead of disabling protection merely to make a query run [27].

A robust ETL pipeline records provenance, validates keys and types, makes missing-value rules explicit, and checks freshness. Avoid treating an Applied Steps sequence as sufficient evidence of quality. The same disciplines that make a dbt or Python pipeline maintainable apply to M even if the editor writes most code.

Takeaway. M's real advantage is the combination of functional transformations, heterogeneous connectors and a managed data preparation interface. It is not a reason to move every database transformation out of PostgreSQL.

6.4 Query folding: from an M expression to a source query

The practical advantage of folding is avoiding unnecessary movement of data. A PostgreSQL database may contain millions of rows, but a report may require only records from 2026 and three columns. Rather than retrieving the whole table and applying a filter in the mashup engine, a connector can translate supported parts of the M expression into a request PostgreSQL understands. PostgreSQL then chooses its own execution plan. Folding and SQL optimization are consequently two different decisions: translating a request and executing that request efficiently [4,7,13].

The following lab uses the supplied PostgreSQL setup script. It assumes the `power_drills.sales` table and its unquoted, lowercase column names. Configure the connection with your own server and database. Credentials should be supplied through the connector, not embedded in M.

Power Query (M)

```
// Navigate to the database relation, not a local CSV
let
    DB = PostgreSQL.Database("localhost", "practice"),
    Source = DB{[Schema="power_drills", Item="sales"]}[Data],
    // These relational steps are candidates for folding
    Filtered = Table.SelectRows(Source,
        each [orderdate] >= #date(2026, 1, 1)),
    // Keep only the fields needed downstream
    Projected = Table.SelectColumns(Filtered,
        {"orderid", "quantity", "unitprice"}),
    // Evaluate the new expression for each retained row
    Added = Table.AddColumn(Projected, "Amount",
        each [quantity] * [unitprice], type number)
in
    Added
```

A plausible equivalent source request is shown below. It is illustrative SQL, not a captured query generated by the connector.

PostgreSQL

```
-- Restrict rows and project only the necessary columns
SELECT orderid, quantity, unitprice,
       quantity * unitprice AS "Amount"
FROM power_drills.sales
WHERE orderdate >= DATE '2026-01-01';
```

The expected result contains orders 5 through 10, with amounts 500, 100, 600, 200, 200 and 400. Their sum is 2,000. Actual output order is not guaranteed without explicit sorting. If all operations fold, the database produces the projected result. If only the predicate and projection fold, the database returns six rows and M computes `Amount`. If nothing folds, the mashup engine may retrieve the entire source and apply all transformations itself. With a ten-row fixture these strategies look similar; with millions of rows the network and memory difference can be substantial.

Execution boundary	Source work	Mashup engine work
Full folding	Predicate, projection, multiplication	Receive result
Partial folding	Predicate and projection	Multiply and return
No folding	Read source rows	Filter, project, multiply

This is a conceptual comparison, not a guarantee for a particular connector version. The optimizer can reorder operations and eliminate work that is not required. A custom function is not necessarily a folding barrier; what matters is whether the engine can translate it in that context. Sorting a large table or buffering it before applying a selective predicate can be expensive. A privacy boundary may also constrain transferring values between sources, preventing an otherwise possible optimization.

`Table.Buffer()` materializes scalar cell values shallowly; nested values are not recursively forced by the operation. It can prevent subsequent folding and increase memory consumption. Use it only after identifying a concrete need. If the goal is merely to prevent downstream folding, `Table.StopFolding()` expresses that intent more directly. Neither operation replaces proper measurement.

6.4.1 How to investigate actual folding

In Desktop, inspect the native query option on applicable steps when the connector exposes it. Its absence does not prove all source work is local. In Power Query Online, folding indicators and query plans are available in supported scenarios. A step indicator describes folding up to that step rather than guaranteeing every later step folds. Query diagnostics, database logs and source-side monitoring can reveal which requests are executed, including repeated preview or metadata requests. Compare the actual refresh duration and returned row volume, not just the time taken by a cached preview.

When using a native SQL query, subsequent folding is not automatic in every connector. Microsoft documents `Value.NativeQuery()` with an `EnableFolding` option for supported sources, including PostgreSQL; validate the specific query and connection [13]. A database view can be a better choice when many reports require the same canonical transformation, especially when it must be versioned, tested and governed centrally.

`dbplyr` translates supported lazy R expressions into SQL and sends a query when needed; `collect()` transfers results into R. Python SQL expression libraries similarly construct source queries, whereas ordinary pandas operations usually act on already retrieved data. A lazy analytical framework may also optimize a local scan without a remote database. These mechanisms share the idea of pushing operations towards the data, but their translation rules and execution boundaries differ.

A nonfolded Import-mode query normally still evaluates the requested transformations against the data available from the connector; it can simply be slow. A nondelegable Power Fx query can operate on only an initial portion of a large source, producing an incorrect result. DirectQuery can impose additional folding requirements or reject unsupported transformations. These are distinct failure modes and should be diagnosed separately.

The absence of a native-query view or folding indicator does not necessarily prove that source pushdown is absent. Connector behavior, diagnostics, and source-side monitoring provide stronger evidence.

7 DAX: semantic models and contextual analytics

7.1 Power BI as a complete analytical system

Power BI is not the name of a single programming language or a database. It combines an authoring environment, data preparation, a reusable analytical model, query execution and a reporting interface. The components can be separated in deployment: several reports can query the same model, and a model can obtain data from several sources. Microsoft documents these relationships and the end-to-end workflow in references [14–16].

Component	What it is	Practical responsibility
Power BI Desktop	Windows authoring application	Connect, transform, model and design reports
Power BI service	Cloud service	Host, share, refresh and manage content
Power Query	Data integration engine and editor	Connect and shape data
M	Expression language	Specify transformation logic
Semantic model	Analytical abstraction and metadata	Define tables, relationships and metrics
DAX	Analytical expression and query language	Calculate results within a model
Report	Interactive pages of visuals	Present and explore analytical results
Dashboard	Service canvas of pinned tiles	Monitor selected information at a glance
Workspace	Service collaboration container	Organize content and access roles
Dataflow	Managed reusable preparation artifact	Share prepared data across consumers
Gateway	Connectivity bridge	Reach sources not directly accessible to the service
Fabric	Integrated cloud analytics platform	Provide storage, engineering and analytical services

Desktop can contain both the model and a report, which is why they are easily confused. In the service they can be separate artifacts. A report stores visual definitions, field bindings, layout and interactions; the model defines what its measures mean. A dashboard in the service is a distinct tile-based artifact, not merely another name for any report page. Dataflows can centralize preparation across multiple models; their storage and orchestration details depend on the dataflow generation and capacity. A gateway is a secure connection mechanism, not another database or a place to write DAX. A workspace organizes content and collaboration, but actual access also depends on permissions, sharing, licensing and capacity. Fabric is relevant when storage or engineering workloads are integrated with the analytical platform; it is not a prerequisite for every ordinary Power BI project.

7.1.1 The three times at which work occurs

```
DESIGN TIME       Desktop: source definition, M, model, report
                          |
                          v
REFRESH TIME      Source -> Power Query -> Import model
                          |
                          v
QUERY TIME        Visual -> DAX query -> model engine -> result
```

In Import mode, refresh reads data, evaluates the preparation pipeline and updates the model's stored contents. A visual interaction sends a query against that model, which generally does not require rerunning M or reading the original database. DirectQuery instead retrieves external data during query execution, subject to caching and source behavior. A hybrid or composite model can combine these patterns. An apparent instantaneous change in a chart is a new analytical result, not evidence that the data has just refreshed.

7.1.2 A practical end-to-end lab

Use the included `Sales`, `Products`, `Customers` and `Calendar` CSV files. The following walkthrough deliberately starts with Import mode, so the data can be explored without a live database or cloud credentials.

Stage	Developer action	What the system does
1. Connect	Get Data; select CSV or PostgreSQL	Establish connector and source schema
2. Retrieve	Select relevant objects	Read preview and source metadata
3. Transform	Open Power Query; select and clean fields	Build M expressions and preview results
4. Type	Set date, whole-number and currency types	Convert values and expose errors
5. Load	Close and Apply	Evaluate queries and populate Import tables
6. Relate	Configure keys in Model view	Establish filter propagation
7. Model	Use dimensions; hide technical keys as appropriate	Expose business-facing fields
8. Calculate	Create measures in DAX	Register reusable expressions
9. Visualize	Add cards, matrix and chart	Generate model queries for visual cells
10. Interact	Add year and product slicers	Change query context and recompute results
11. Publish	Publish to an authorized workspace	Create or update service artifacts
12. Operate	Configure access, credentials and refresh	Enforce access and refresh configuration
13. Maintain	Inspect timings and validate results	Expose diagnostics and execution behavior

At stages 1–4, ensure each input has the expected schema and grain. Use M to create a `Revenue` column from `Quantity` and `UnitPrice` if that column is not already present. A preview is sampled or cached in some cases, so verify the loaded row counts after applying changes. In stage 5 the imported model should contain ten order rows. A refresh later evaluates the stored query definitions; editing a source CSV does not automatically update the already imported data.

At stages 6–8 establish the three one-to-many relationships described in the practice model and define `Total Revenue = SUM(Sales[Revenue])`. Create a matrix with `Products[ProductName]` as rows and the measure as values. The unfiltered result should be 1,300, 450 and 1,200 for Keyboard, Mouse and Monitor respectively. Add `Calendar[Year]` as a slicer. For 2026 the values become 700, 300 and 1,000. The formula stays the same; the model query supplies different filters.

At stage 9 use a card for total revenue and a monthly chart with `Calendar` fields. At stage 10 select a product in the chart and observe which other visuals change; interactions can be configured to filter, highlight or not interact. For stage 11 you need an appropriate account and workspace permission. For stage 12 configure source credentials and a gateway if the service cannot directly reach the source; a local file path may not be accessible to the service without suitable connectivity. Check refresh history, permissions and model security rather than assuming publishing handles them automatically. Sharing and capacity rules vary, so confirm current licensing and tenant policies before deployment. Finally, use Performance Analyzer and reconciliation checks to validate the solution after changes.

7.2 What a semantic model actually contains

A semantic model is a reusable description of how data should be interpreted and queried. It includes table definitions and storage configuration, relationships, columns and types, calculation definitions, formatting, hierarchies, and potentially security rules. It can store compressed data in Import mode or describe access to external data. Its key contribution is shared business meaning: one definition of revenue, one approved date dimension and one set of relationships can serve many visuals and reports.

Familiar object	Similarity to a semantic model	Important difference
PostgreSQL schema	Names, types and relations	Primarily structures database objects
View	Reusable logical definition	Usually returns a relation, not contextual metrics
Materialized view	Precomputed data	Refreshes stored query output rather than defining all model behavior
Foreign key	Declared relationship between entities	Enforces integrity; model relationship primarily controls analytical navigation and filtering
Data frame	Tabular values and column names	Does not natively define a reusable cross-table metrics layer
Warehouse	Curated analytical storage	May serve many models and tools; not itself a Power BI report
Report	Consumes model fields	Layout and presentation rather than the authoritative metric definition

A PostgreSQL view can encode a business metric, and an analytical application can layer filter-aware functions over it. That is a useful analogue to a semantic model. However, a DAX measure is designed to be evaluated under the current analytical context across relationships. A SQL foreign key and a Power BI relationship are not equivalent: the database constraint protects referential integrity, while the model relationship determines filter propagation. Validate keys before loading rather than expecting a model to repair invalid facts.

Grain means exactly what one row represents. In the supplied model one `Sales` row represents one order; if an order could contain several lines, `COUNTROWS(Sales)` would instead count lines. A distinct count of `OrderID` would then be needed to count orders. Dimension tables describe entities used for grouping and filtering, while facts record events or measurements. A star schema simplifies the path from business categories to underlying facts and helps avoid double counting.

7.2.1 Same data, different model behavior

Create the `Total Revenue` measure once. With an active `Products`-to-`Sales` relationship, selecting Keyboard restricts `Sales` to orders 1, 4, 5 and 9. Their revenue is 200 + 400 + 500 + 200 = 1,300. If the `Products` table is disconnected, the same slicer does not filter `Sales` and the card still displays 2,950. If the `Calendar` relationship is absent, a year slicer may likewise leave the total unchanged. These differences arise from metadata and propagation, not changed arithmetic.

Model configuration	Keyboard selection	Keyboard and 2026
Both active relationships	1,300	700
`Products` disconnected	2,950	2,000
`Calendar` disconnected	1,300	1,300

This makes model testing as important as testing formulas. Cardinality has two related meanings: the number of unique values in a column, and the multiplicity of a relationship. A product key should be unique on the one side. A fact table can contain many records with that key. Single-direction filtering from dimensions to facts gives predictable analytical behavior. Many-to-many or bidirectional relationships are sometimes appropriate but require careful examination of ambiguous filter paths and totals.

Reusable semantic models support consistent calculations across reports. A governed measure can define margin as total profit divided by total revenue instead of relying on each report author to derive it independently. Row-level security restricts which rows a consumer can query; object-level security can restrict access to model objects. These mechanisms are not substitutes for correctly configured workspace roles, source permissions or data minimization. A visual hiding a column is not a security control.

In Power BI workspaces, RLS applies to Viewer-role consumers rather than Admins, Members or Contributors with editing access. Test using the intended consumer identity and permissions [28].

7.3 Storage modes and the two analytical engines

In a row-oriented database, data for a record is typically stored together; in a columnar engine, values of a column are organized together. Columnar storage is useful for scanning a subset of fields over many rows, such as calculating the sum of revenue by product. Repeated values may compress efficiently, while unique identifiers or long text often cost more space. Compression effectiveness is not determined by cardinality alone, but high cardinality is a useful signal to investigate.

VertiPaq is the compressed columnar storage engine used in Import models and used for analytical processing with Direct Lake. The Formula Engine interprets DAX and orchestrates computation; the Storage Engine handles supported scans, grouping and aggregation. A query may require multiple storage requests.

DirectQuery storage-engine mechanisms issue source-native requests; the external database is the source, not itself the DAX Storage Engine. It is not accurate to equate every DAX iterator with a slow interpreted loop: engines can optimize expressions and push supported work into storage operations [7,8,15].

Mode or configuration	Query-time behavior	Practical trade-off
Import	Query compressed model data	Fast interactive exploration; refresh governs freshness
DirectQuery	Request data from external source	Latency, source load and feature restrictions
Dual	Use cache or DirectQuery as appropriate	More complex model design and consistency decisions
Composite model	Combine supported sources or storage modes	Flexible integration; cross-source behavior needs testing
Direct Lake	Access supported OneLake-backed tables	Fabric dependencies, capacity and configuration considerations

In Import mode the database is often busiest during refresh rather than during each interaction. In DirectQuery, report consumers can generate external queries repeatedly, so source indexing, workload concurrency and gateway latency become part of the report's performance. DirectQuery does not guarantee that each visual reads the very latest source value because caching and query behavior matter. Composite models can mix storage approaches; Dual dimension tables may serve queries from a cache or participate in source queries depending on the plan. Direct

Lake accesses supported data in OneLake and has variant-specific capabilities and fallback rules; consult the current documentation for the chosen configuration [15].

To diagnose a slow report, first identify the slow visual. Then separate visual rendering time from DAX query time, inspect the query and determine whether storage scans, Formula Engine work or source latency dominate. In Import mode, reduce unused columns and investigate high-cardinality fields before redesigning every measure. In DirectQuery, inspect the generated source requests and database query plans. Test the result under realistic concurrency and filter selections. Performance is a system property, not a simple ranking of languages.

7.4 Build a valid star schema

A fact table has a declared grain: here, one order per row. Dimension keys must be unique, and facts reference those keys. Relationships enable predictable filter propagation. Avoid duplicating fact data through accidental many-to-many joins.

```
Products (1)  --------> (*) Sales (*) <--------  (1) Customers
                              ^
                              |
                             (1)
                          Calendar
```

The `Calendar` dimension is a contiguous daily sequence, not a distinct list of observed transaction dates. Create it in DAX if you do not import the supplied CSV:

DAX

```
Calendar =
// Create a contiguous date dimension and derive grouping attributes
ADDCOLUMNS (
    CALENDAR (DATE (2025,1,1), DATE (2026,12,31)),
    "Year", YEAR ([Date]),
    "MonthNumber", MONTH ([Date]),
    "YearMonth", FORMAT ([Date], "yyyy-MM")
)
```

Create active single-direction one-to-many relationships from `Products`, `Customers` and `Calendar` to `Sales`. Mark `Calendar` as a date table for classic date-column time intelligence. Use `Calendar` fields in slicers, not separate automatic date tables. DAX measures should live in a documented semantic model with meaningful names and descriptions.

7.5 Measures, calculated columns and queries

Calculation	When evaluated in Import model	Result	Example
M custom column	Refresh / query load	Stored value per row	`Quantity` times `UnitPrice`
DAX calculated column	Model refresh	Stored value per row	`Revenue` tier
DAX calculated table	Model refresh	Stored table	`Calendar`
DAX measure	On demand	Scalar per filter context	Total `Revenue`
DAX query	Query submission	Result table	`Revenue` by product
DAX visual calculation	Visual evaluation	Value over visual data	Running sum in visual

Static derived columns can usually be created upstream in PostgreSQL or M. DAX columns are useful when calculation needs model relationships or model-specific attributes, but they increase storage. Measures are usually the correct choice for context-dependent totals and ratios. A calculated column can be filtered by a slicer but is not recomputed by the slicer. The timing shown is for Import models; DirectQuery has additional restrictions [5,7].

A visual calculation is stored with the visual rather than the semantic model and operates on the visual result at its displayed grain. For example, `RUNNINGSUM([Total Revenue])` can accumulate revenue along a visual axis. It is not a reusable model measure or a per-row stored column; relationship-dependent functions and supported operations differ [29].

DAX

```
Total Quantity =
// Sum visible quantities under current report filters
SUM (Sales[Quantity])
```

DAX

```
Total Orders =
// Count fact rows at the declared order grain
COUNTROWS (Sales)
```

DAX

```
Total Cost =
// Iterate fact rows and retrieve cost through the product relationship
SUMX (
    Sales,
    // The related cost comes from the product dimension
    Sales[Quantity] * RELATED (Products[UnitCost])
)
```

DAX

```
Gross Profit =
// Subtract contextual cost from contextual revenue
[Total Revenue] - [Total Cost]
```

DAX

```
Profit Margin =
// Recalculate the ratio in each filter context, including totals
DIVIDE ([Gross Profit], [Total Revenue])
```

Each definition is a separate measure. Unfiltered results: revenue 2,950; quantity 28; orders 10; cost 1,770; profit 1,180; margin 40%. Use the same measures in cards and matrices. They should not need to be rewritten for each product, customer or year.

7.6 DAX evaluation contexts and context transition

Context or operation	Definition	Example
Row context	Current row during column or iterator evaluation	`Sales[Quantity]` inside `SUMX()`
Filter context	Allowed values and propagated filters during evaluation	Product and Year slicers
Context transition	Conversion of existing row context to filters	`CALCULATE()` inside a calculated column

SQL programmers often assume that a current row implicitly filters every related table. It does not. Row context provides values for the current row; filter context governs which rows are visible to an aggregation. In DAX, `CALCULATE()` bridges these contexts. An invoked measure within row context performs context transition automatically, which can be powerful but also expensive when applied over a large iterator [5,6,8].

Context transition is an operation, not a third context. It creates filters from the values in active row contexts, not a physical row pointer; duplicate rows with identical values need not be uniquely identified.

In the `Products` table, compare these two calculated columns:

DAX

```
Revenue A =
// A calculated column has row context, but SUM receives no product filter
SUM (Sales[Revenue])
```

DAX

```
Revenue B =
// Context transition makes the current product row a filter
CALCULATE (SUM (Sales[Revenue]))
```

Product	Revenue A	Revenue B
Keyboard	2,950	1,300
Mouse	2,950	450
Monitor	2,950	1,200

The first `SUM()` sees no filter on `Products` from the current row alone. The second converts the row into filters, which propagate to `Sales`. In production, avoid storing a redundant total for each product unless a specific model requirement justifies it. The exercise exists to expose context transition.

7.6.1 Trace row context and filter context by hand

A row context supplies values from the current row. In the first three `Sales` rows, the expression `Sales[Quantity] * Sales[UnitPrice]` evaluates to 200, 150 and 200. `SUMX()` establishes a row context for each visible `Sales` row and sums the expression values. A calculated column also has a row context, but stores its individual row result at refresh in an Import model. An iterator inside another iterator can have multiple active row contexts; use variables to capture outer values and avoid confusing the current row of one table with another.

Filter context is different: it describes which values and rows are permitted during a calculation. A visual matrix creates a filter for each grouping cell, a slicer adds selected values and the model propagates filters through active relationships. The intersection of these restrictions determines which fact rows are available. For example, a 2026 slicer and a Keyboard matrix row together allow orders 5 and 9, resulting in 700. A card with only the 2026 slicer allows all six 2026 orders, resulting in 2,000. The total cell has its own context and is not necessarily an arithmetic sum of the visible cells.

Filter context	Visible order IDs	Total Revenue
No ordinary report filters	1–10	2,950
Keyboard	1, 4, 5, 9	1,300
Year 2026	5–10	2,000
Keyboard and 2026	5, 9	700
Mouse and Costa Rica	8	200

SQL can reproduce these results using `WHERE` and joins, or grouped aggregates; a pandas application can explicitly filter a frame and call `sum()`. The difference is that DAX receives a filter context from the surrounding model query. You do not have to rewrite the measure for each visual cell. In a PostgreSQL application, a metrics layer or query generator can supply similar behavior, but that infrastructure must be provided explicitly.

7.6.2 Why context transition exists

Revisit `Revenue A` and `Revenue B` above. In the `Products` calculated column, the current row is Keyboard. That row context allows you to read `Products[ProductName]`; it does not by itself restrict the fact table for an ordinary `SUM()` expression. Thus `Revenue A` returns the overall 2,950. In `Revenue B`, `CALCULATE()` converts the current product row into filters on the model. The relationship propagates the product restriction to `Sales`. Now the same aggregation sees only Keyboard orders and returns 1,300. The distinction is not whether the program knows the current product; it is whether the aggregation receives a filter that represents it.

A measure reference in a row context implicitly causes context transition. For example, iterate the visible customer IDs and evaluate the existing measure separately for each customer:

DAX

```
Average Customer Revenue =
// Iterate customer keys and evaluate the measure for each key
AVERAGEX (VALUES (Customers[CustomerID]), [Total Revenue])
```

The three unfiltered customer totals are 1,100, 450 and 1,400; the average is 983.33. Replacing the measure reference with the naked expression `SUM(Sales[Revenue])` removes that implicit transition and can repeat the outer-context total for every iteration. `CALCULATE(SUM(Sales[Revenue]))` restores it. This is a useful debugging test because it isolates the effect of context transition without changing the arithmetic.

`VAR` does not store a formula to be re-evaluated whenever a context changes. A variable's result belongs to its defining context. To obtain an all-products denominator, define the variable using `CALCULATE([Total Revenue], REMOVEFILTERS(Products))`; do not first store `[Total Revenue]` and expect a later `CALCULATE()` around that scalar to recalculate it.

Measure evaluations performed inside row context trigger automatic context transition. This behavior is central to understanding iterators such as `AVERAGEX()`, `SUMX()`, and other `X`-functions.

7.7 `CALCULATE()` and filter manipulation

DAX

```
Accessories Revenue =
// Replace the existing filter on Category; preserve other dimensions
CALCULATE ([Total Revenue], Products[Category] = "Accessories")
```

DAX

```
Accessories Keep =
// Intersect with the existing category filter rather than replacing it
CALCULATE (
    [Total Revenue],
    // Intersect with, rather than overwrite, the current category filter
    KEEPFILTERS (Products[Category] = "Accessories")
)
```

Category filter	Base revenue	Accessories `Revenue`	Accessories Keep
No filter	2,950	1,750	1,750
Accessories	1,750	1,750	1,750
Electronics	1,200	1,750	`BLANK()`

The plain Boolean argument replaces a pre-existing filter on the same column; `KEEPFILTERS()` intersects it. Other filters (such as year or customer) remain. This is not a global bypass of security: filter modifiers do not remove row-level security. The example assumes the category slicer is the only product filter.

Function	Use	Technical caveat
CALCULATE()	Change context of scalar	Can perform context transition
CALCULATETABLE()	Change context of table	Returns a table, not a measure scalar
FILTER()	Row-wise predicate over a table	Prefer simpler Boolean filters when equivalent
REMOVEFILTERS()	Clear selected filters	Does not return a table
ALL	Clear filters or return unfiltered table	Auto-exist and expanded-table semantics matter
ALLEXCEPT()	Preserve designated columns	Does not infer an absent grouping filter
ALLSELECTED()	Visual totals under outer selections	Shadow filter contexts can complicate iterators
VALUES()	Distinct contextual values	May include unknown blank dimension member
ISINSCOPE()	Detect grouping level	Not equivalent to HASONEVALUE()

ALLSELECTED() is not simply a function for fetching slicer values. It restores an outer query context in typical visual calculations; nesting it inside iterators can produce surprising results. Test with the actual generated visual query and avoid broad filter removal when only one column should be cleared. Auto-exist can merge filters on columns of the same table; the resulting set may not be the Cartesian product of separately selected values. Prefer clear dimension design and explicitly test combinations [5,8].

DAX

```
Product Share =
// Remove product filters only; preserve other dimensions
DIVIDE (
    [Total Revenue],
    // Re-evaluate the denominator without product restrictions
    CALCULATE ([Total Revenue], REMOVEFILTERS (Products))
)
```

DAX

```
Selected Product Share =
// Use the outer product selection as the denominator
DIVIDE (
    [Total Revenue],
    // Retain the outer product selection for the denominator
    CALCULATE ([Total Revenue], ALLSELECTED (Products[ProductName]))
)
```

If a report selects Keyboard and Monitor only, the first share uses 2,950 as denominator (44.07% and 40.68%), while the second uses 2,500 (52% and 48%), assuming ProductName is the only product slicer and grouping column. Country and year filters remain active in both measures.

7.7.1 Reading filter functions as operations on the evaluation environment

CALCULATE() evaluates a scalar under a modified context, while CALCULATETABLE() evaluates a table expression under a modified context. A Boolean filter on Products[Category] replaces an existing filter on that column by default. KEEPFILTERS() changes this behavior to intersection. Other columns and dimensions can still constrain the result. FILTER() evaluates a predicate row by row over a supplied table and returns qualifying rows; it does not by itself alter the surrounding evaluation context. It can be used as a filter argument to CALCULATE() where a

more complex predicate is needed. The evaluation of filter arguments and context transition has subtleties in nested expressions; do not assume every argument executes sequentially in the textual order [5,8].

Conceptually, explicit filter arguments are evaluated in the incoming context, context transition produces filters from active row contexts, modifiers adjust the resulting context, and the evaluated explicit filters are applied before the main expression is evaluated. A filter argument does not automatically receive the context transition of its enclosing CALCULATE. For example, inside SUMX over Products, use the current row ProductID directly in a filter argument rather than assuming SELECTEDVALUE sees the transitioned filter [30].

`REMOVEFILTERS()` is a clear choice when the intent is to remove a restriction. `ALL()` can also return a table and thus supply rows to an iterator. Removing a filter on `ProductName` is narrower than removing filters from the entire `Products` table. A category filter may survive the former but not the latter. `ALLEXCEPT()` removes other filters from a table while preserving filters on designated columns; it does not manufacture a category filter if none was present. These operations can involve expanded-table semantics, so test relationships and scope when clearing whole tables.

An expanded Sales table includes columns reached through regular many-to-one relationships to its dimensions. Removing filters from Sales as a table modifier can therefore remove dimension filters, not merely fact-column restrictions. For example, in the fixture CALCULATE([Total Revenue], REMOVEFILTERS(Sales)) can remove a Products category filter; removing Sales[OrderID] alone does not. Expanded tables are a semantic concept, not physical joins materialized for every query; limited relationships do not participate in the same expansion [31].

`ALLSELECTED()` is useful when the denominator should follow outer selections rather than the current visual row. The denominator for a Keyboard share can be all products (2,950), all products currently selected (2,500 when Keyboard and Monitor are selected), or all products in the current category (1,750 for Accessories). There is no universal percentage denominator. The metric definition must choose one intentionally.

`VALUES()` retrieves contextual distinct values and may expose an unknown blank member for unmatched relationships. `SELECTEDVALUE()` retrieves a single value only when exactly one distinct value is visible, otherwise returning its alternative or blank. `ISINSCOPE()` checks whether a column participates in the relevant grouping scope; multiple hierarchy columns can be in scope, so test the deepest intended level first. A slicer can select one category without that column being in the visual grouping scope. `HASONEVALUE()` tests the count of distinct values, not grouping scope. For subtotals and hierarchies, that difference is essential.

As a practical exercise, define a dynamic label and put it in a card next to the matrix:

DAX

```
Selected Product Label =
// The alternative covers zero or multiple distinct visible names
SELECTEDVALUE (Products[ProductName], "Multiple or none")
```

Without a product filter, the label shows the alternative. With Keyboard selected, it shows Keyboard. If the visual groups by `ProductName`, each detail cell may see a single product, while the total sees multiple products. That is why context-sensitive expressions should be tested at detail rows, subtotals, totals and under external slicers.

7.7.2 Choosing a calculation's location

The choice between a Power Query derived column and a DAX measure depends on the grain and evaluation time. `Revenue = Quantity * UnitPrice` is a row-level property that can be computed in PostgreSQL, M or a DAX calculated column. A margin measure, however, divides totals under an arbitrary filter context. Precomputing a margin per row and averaging it gives each row equal weight; the correct aggregate margin is generally weighted by revenue. A model measure can reuse the numerator and denominator across reports without adding a repeated result to every fact row.

In Import models, calculated columns and calculated tables add stored values to the model and are ordinarily processed during refresh. Measures are definitions evaluated on demand; their results may be cached but are not stored as one fixed value per fact row. A programmer accustomed to pandas or `dplyr::mutate()` may instinctively create a column for every derived value. Ask whether the value belongs to an individual observation or depends on a user's analytical selection. For context-dependent ratios, counts and comparisons, a measure is often the right abstraction. DirectQuery and other storage modes have different restrictions and execution behavior.

7.8 Virtual tables, lineage and relationships

DAX

```
Average Customer Revenue =
// Iterate visible customer IDs; invoking the measure transitions context
AVERAGEX (VALUES (Customers[CustomerID]), [Total Revenue])
```

Unfiltered result: 983.33. `VALUES()` produces a contextual table of customer identifiers, and the measure invocation transitions each row into filter context. This differs from the average order revenue (295): the grain is customer, not transaction. Virtual tables need not be materialized as persistent model tables.

AVERAGEX excludes blank expression results. If a fourth registered customer has no sales, the original measure remains 983.33, whereas averaging over all four visible registered customers with no purchases treated as zero gives 737.50:

DAX

```
Average Revenue per Registered Customer =
AVERAGEX (VALUES (Customers[CustomerID]),
    COALESCE ([Total Revenue], 0))
```

Choose the population intentionally; with unmatched foreign keys, also decide whether to exclude the relationship-generated unknown member.

Data lineage is the association between a virtual column and its originating model column. It allows filters generated from a virtual table to affect the model appropriately. Not all transformations retain lineage; a column constructed with an arbitrary string expression may not propagate filters as expected. `TREATAS()` explicitly maps values onto a target column and is useful for disconnected slicers. It is not a substitute for well-designed physical relationships.

For example, SELECTCOLUMNS(Products, “Key”, Products[ProductID]) preserves the lineage of the direct column reference, whereas concatenating an empty string to the reference produces a computed text expression without that lineage. Equal displayed keys need not have equal filtering behavior. TREATAS can explicitly apply the computed keys to Products[ProductID] [32].

Function	Purpose	Example situation
`RELATED()`	Fetch one related value in row context	Unit cost from `Products`
`RELATEDTABLE()`	Obtain related fact rows	Customer order count
`LOOKUPVALUE()`	Value by matching criteria	Lookup without suitable relationship
`USERELATIONSHIP()`	Use an existing alternative relationship	Revenue by shipping date
`CROSSFILTER()`	Change propagation for one calculation	Count customers buying selected products
`TREATAS()`	Apply virtual filters	Disconnected scenario selector

For `USERELATIONSHIP()`, add a `ShipDate` column equal to `OrderDate` plus three days and create an inactive `Calendar`-to-`ShipDate` relationship. Use `CALCULATE([Total Revenue], USERELATIONSHIP(Sales[ShipDate], Calendar[Date]))`. This temporarily selects the alternative date relationship for the measure. If the model must support simultaneous independent order and ship date slicers, role-playing date dimensions may be more appropriate. `CROSSFILTER()` should be used carefully to avoid ambiguous paths; security rules impose additional constraints.

Alternative date measure

```
// Use the existing inactive shipping-date relationship for this measure
CALCULATE ([Total Revenue],
    USERELATIONSHIP (Sales[ShipDate], Calendar[Date]))
```

For `TREATAS()`, create a disconnected `Country Selector = DISTINCT(Customers[Country])` table, then:

DAX

```
Scenario Revenue =
// Map disconnected selector values onto the customer column
CALCULATE ([Total Revenue],
    TREATAS (VALUES ('Country Selector'[Country]), Customers[Country]))
```

Selecting Costa Rica yields 1,100 without a physical relationship. The disconnected selector is useful for a scenario or comparison; document how it interacts with real Customer filters.

7.9 Ratios, time and non-additive measures

Metric	Aggregation property	Correct total strategy
`Revenue`	Additive	Sum transaction revenue
Inventory balance	Semi-additive	Sum entities at a valid snapshot time
Profit margin	Non-additive	Divide total profit by total revenue
Unique customers	Non-additive	Recompute distinct count
Average selling price per unit	Non-additive	Divide revenue by quantity

A matrix total is a fresh measure evaluation in a broader filter context, not necessarily the sum of cells. If two categories have profit margins 20% and 80%, averaging those percentages gives 50% even when revenue weights differ dramatically. Use `DIVIDE()` of summed profit and revenue instead. Inventory snapshots require choosing the latest relevant observation per entity, potentially different dates for different entities.

The fixture illustrates three different price-related measures: average order revenue is $2950/10 = 295$; weighted selling price per unit is $2950/28 = 105.36$; mean order-level UnitPrice is 115. These denominators answer different questions.

DAX

```
Revenue YTD =
// Accumulate dates within the current calendar year
TOTALYTD ([Total Revenue], Calendar[Date])
```

DAX

```
Revenue PY =
// Shift the current date selection to the prior year
CALCULATE ([Total Revenue],
    SAMEPERIODLASTYEAR (Calendar[Date]))
```

DAX

```
Revenue YoY =
// Capture current and prior values before calculating growth
VAR CurrentValue = [Total Revenue]
// Evaluate the prior-year measure before forming the ratio
VAR PreviousValue = [Revenue PY]
RETURN
    // DIVIDE returns blank when the prior-year denominator is zero
    DIVIDE (CurrentValue - PreviousValue, PreviousValue)
```

DAX

```
Rolling 3M =
// Use the latest visible date as the window endpoint
CALCULATE (
    [Total Revenue],
    // Anchor the trailing interval at the latest visible date
    DATESINPERIOD (Calendar[Date], MAX (Calendar[Date]), -3, MONTH)
)
```

2026 month	Revenue	YTD	Prior-year month	YoY
January	500	500	200	150%
February	100	600	150	-33.33%
March	600	1,200	200	200%
April	200	1,400	400	-50%
May	200	1,600	BLANK()	BLANK()
June	400	2,000	BLANK()	BLANK()

These examples use classic date-column time intelligence. A rolling window uses the latest date in the current context, which may be later than the latest actual transaction. For a real dashboard explicitly choose an as-of date and define whether incomplete periods are included. Calendar-based time intelligence and host features evolve; the classic formulas avoid depending on preview capabilities [6]. In PostgreSQL, a window sum over a complete monthly calendar and explicit frame can implement analogous behavior; in R use a complete time series and cumulative or rolling functions, and in pandas use resampling and rolling windows. In all cases gaps and incomplete periods must be handled deliberately.

7.9.1 Make period comparisons explicit

A date table contains a row for every day in the reporting range, including dates without transactions. It provides stable month, quarter and year groupings; it also lets time-intelligence functions shift or expand a date filter without depending on which days happen to contain sales. For classic time intelligence, use a marked, contiguous date table with an active relationship to the relevant fact date. Model calendar or fiscal-year rules explicitly.

Consider March 2026. A month filter initially allows March dates only. A YTD calculation expands that date window to January through March and produces 1,200. A prior-year comparison shifts the March period to March 2025 and returns 200. The March 2026 revenue of 600 therefore represents 200% growth. A trailing three-month window ending in April 2026 includes February, March and April (100 + 600 + 200 = 900). A running total across years includes all earlier transactions, not just those within the current year. These calculations are all aggregations over different date sets.

Period	Monthly revenue	YTD revenue	Prior-year month
Jan 2026	500	500	200
Feb 2026	100	600	150

Period	Monthly revenue	YTD revenue	Prior-year month
Mar 2026	600	1,200	200
Apr 2026	200	1,400	400
May 2026	200	1,600	Blank
Jun 2026	400	2,000	Blank

The fixture has no sales after June 2026 and none after April 2025. Missing prior-period revenue is not proof of zero business activity; the data may simply be incomplete. Define whether to compare complete months, matched year-to-date periods or explicitly selected date ranges. A report using an entire date dimension can show future empty dates, so use a documented as-of date where appropriate.

Inventory balances are semi-additive: summing balances across products on a given date can make sense, but summing daily ending balances across a month usually does not. A distinct count is non-additive because the same customer can appear in several categories. Ratios are non-additive because their denominators differ. An average selling price is total revenue divided by total quantity, not an unweighted average of row prices. In this fixture the weighted price is 2,950 / 28 = 105.36. These are metric-definition choices, not merely DAX syntax.

7.10 DAX query view, ranking and statistics

DAX

```dax
// Return a query result table using existing model relationships
EVALUATE
SUMMARIZECOLUMNS (
    Products[ProductName],
    "Revenue", [Total Revenue],
    "Orders", [Total Orders]
)
ORDER BY [Revenue] DESC
```

Product	Revenue	Orders
Keyboard	1,300	4
Monitor	1,200	3
Mouse	450	3

This query depends on the model's relationships rather than an explicit join. DAX Query View supports testing table expressions and temporary measures using DEFINE MEASURE. Unlike SQL UPDATE, EVALUATE does not modify persisted source rows. Use query-level tests to compare expected KPI outputs under multiple filter combinations.

DAX

```dax
Product Rank =
// Rank within the selected product universe; suppress the total rank
IF (ISINSCOPE (Products[ProductName]),
    RANKX (ALLSELECTED (Products[ProductName]),
        [Total Revenue], , DESC, DENSE))
```

Ranks: Keyboard 1, Monitor 2, Mouse 3. Modern DAX also offers window-oriented functions such as WINDOW, OFFSET, INDEX and RANK. They require careful relation, ordering and partition definitions, comparable in intent to SQL window functions but not interchangeable in context semantics. A row offset over months with missing observations should use a proper Calendar dimension rather than rely on fact rows.

Statistical task	DAX	R / Python / PostgreSQL
Median	`MEDIAN`	median / Series.median / percentile_cont
Sample SD	`STDEV.S`	sd / Series.std / stddev_samp
90th percentile	`PERCENTILE.INC`	quantile / quantile / percentile_cont
Linear fit	`LINESTX`	lm / statsmodels or sklearn / regr functions
Causal inference	No comprehensive native framework	R/Python specialized libraries

DAX can produce regression summaries, but an analytical measure is not a complete statistical workflow. Inference requires assumptions, diagnostics, study design and uncertainty estimation. The fixture's 90th percentile is 510. Use statistical languages for substantive modeling and publish validated results to the semantic model as appropriate.

7.11 Performance practice

Begin with the simplest correct measure. Use Performance Analyzer to identify slow visuals, DAX Query View to reproduce the calculation, and DAX Studio where available to inspect engine timings and query plans. Distinguish expensive source execution, high-cardinality visual grouping, Formula Engine work and repeated Storage Engine requests. Reduce model cardinality, remove unused columns, use meaningful dimension keys, and avoid unnecessary calculated tables. Performance is empirical, not determined solely by whether an expression contains `SUMX()` or `FILTER()` [7,8].

For repeatable benchmarks, record model version, storage mode, cardinalities, filter selection, capacity, source latency and concurrency. Separate cold-cache from warm-cache runs; repeat the same query and report median and spread rather than a single best timing. Separate Formula Engine and Storage Engine timings from network and visual rendering costs. Cache clearing is environment-dependent and should not disrupt shared production workloads.

Takeaway. The genuinely new skills are dimensional modeling, evaluation context, context transition, relationship semantics and the analytical engine. These are more valuable than memorizing a long list of DAX functions.

8 Architecture and engineering decisions

8.1 Choose the right execution layer

Problem	Preferred starting point	Reason and caveat
Shared relational cleanup	PostgreSQL views or ETL	Reuse centrally; retain constraints and query optimization
Department-owned workbook consolidation	M	Accessible repeatable shaping; monitor source drift
Interactive ratio across report filters	DAX	Context-aware metric, not fixed aggregate
Approval or form workflow	Power Fx + managed service	Fast delivery if connector/security fit
Specialized statistical model	R or Python	Rich diagnostics, validation and modeling libraries
Large-scale API integration	Python or managed ETL	Explicit retries, monitoring and orchestration
Complex transactional workflow	PostgreSQL + application service	Enforce consistency, authorization and transactions
Embedded report calculations	DAX	Reusable semantic definitions

There is no requirement to use every layer. If a company already has a reliable PostgreSQL transformation layer, recreating it in M adds maintenance and may introduce inconsistent metrics. If a calculation depends on an interactive slicer, precomputing it in M usually fails to capture the dynamic context. If an application needs custom protocols, complex multi-record transactions or extensive automated tests, conventional application engineering may be a better fit than a collection of low-code formulas.

8.2 Power Apps versus conventional application frameworks

Dimension	Power Apps / Power Fx	Shiny / Streamlit / conventional Python
UI development	Managed visual editor and formulas	Code and framework components
Reactivity	Property dependencies	Shiny dependency graph; Streamlit rerun model
Identity and connectors	Managed integration, configuration dependent	Explicit implementation or managed hosting
Custom logic	Host constraints	Broad library and runtime control
Deployment	Environment and solution lifecycle	CI/CD, containers or managed hosting
Portability	Power Platform dependence	Higher potential portability, not automatic
Cost	Licensing and connector choices	Infrastructure, engineering and support costs

A managed app can reduce the time to deploy routine internal forms, especially where Microsoft identity and existing governance are available. Conventional frameworks provide more control over source, tests, interfaces and deployment. Neither guarantees good software. Low-code applications still need requirements, validation, security, error handling, maintainability and ownership. Shiny and Streamlit are not interchangeable either: Shiny models reactive dependencies explicitly while Streamlit typically reruns script logic on interaction. Choose the architecture based on the team's actual capabilities and service requirements, not only the initial development effort.

8.3 Engineering lifecycle comparison

Concern	Power Fx / Power Apps	M / Power Query	DAX / Power BI
Version control	Solutions and source export where supported	Query text and project artifacts	PBIP/TMDL and model definitions
Testing	Formula and UI tests; connector integration	Fixture tables, schema and row-count tests	Query assertions under filter contexts
Debugging	Monitor and app diagnostics	Query diagnostics and step inspection	Performance Analyzer, Query View, DAX Studio
Dependencies	Connections, components, solutions	Source queries, connectors, functions	Measures, model relationships, metadata
Deployment	Dev/test/prod environments and solutions	Dataflows or semantic model deployment	Deployment pipelines or CI/CD where available
Governance	Data policies, roles, connector controls	Privacy levels, credentials, gateways	RLS/OLS, workspace roles, permissions

These capabilities belong partly to the hosting products, not the languages. Source control and ALM support vary by product version, artifact type, environment and licensing. Review current official documentation before standardizing a deployment workflow. Exporting an app or report is not equivalent to obtaining a clean, independently buildable text repository; verify that the intended source format supports meaningful diffs and reproducible deployment.

Concern	PostgreSQL	R	Python
Version control	SQL migrations, schema and views	Git, source and Quarto	Git, modules and configs
Testing	SQL assertions, pgTAP optional	testthat	pytest
Profiling	EXPLAIN ANALYZE, statistics	profvis, Rprof	cProfile, line profilers
Dependencies	Server versions, extensions	renv and package libraries	Lockfiles and virtual environments
CI/CD	Migration tests, rollback planning	Test and render pipelines	Build, test, deploy pipelines
Security	Roles, constraints, RLS	Runtime/service identity	Runtime/service identity

PostgreSQL migrations and schema constraints provide a strong foundation for reproducible database changes. R and Python package lockfiles improve reproducibility but do not pin all system dependencies, hardware or external services. Comparable discipline in Microsoft tools involves documenting connectors, environment variables, permission sets, refresh settings and semantic-model definitions. Use service accounts or managed identities where appropriate and avoid embedding personal credentials.

8.3.1 A minimum quality contract for citizen development

A low-code solution should have an identified owner, documented business rules, approved data access, validation, a test fixture, recoverable deployment, a support pathway and a retirement plan. Classify projects by risk: a personal productivity experiment can tolerate a lighter process than a tool that affects disbursements, personnel records or official institutional decisions. Escalate when an app crosses departmental boundaries, handles sensitive data, requires strong transactional guarantees, or has become operationally critical. Technical debt accumulates when unsupported spreadsheets and apps become invisible dependencies.

Takeaway. Evaluate total cost of ownership, not just initial implementation speed. Governance enables wider adoption by making experimentation safe and supportable.

9 Integrated case study: AI-assisted vocabulary learning

9.1 Requirements and data model

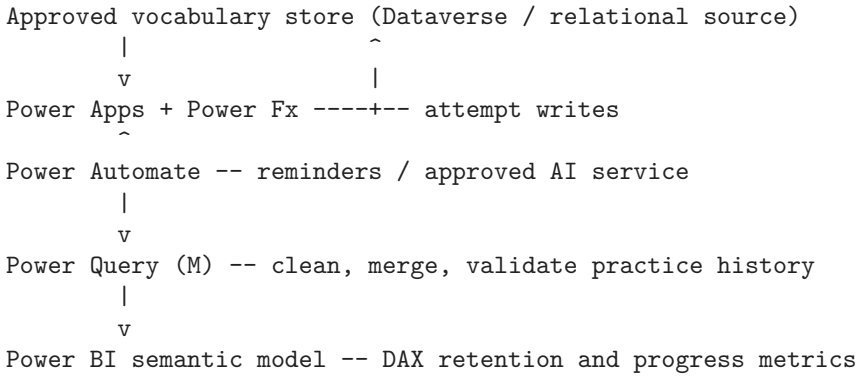
The illustrative product maintains a bilingual vocabulary repository, enriches it with AI-generated examples, schedules practice, delivers interactive quizzes and monitors learning. The relational grain matters: Words stores one record per vocabulary item and Attempts stores one record per submitted response. A unique attempt ID supports idempotent ingestion; timestamps are stored in UTC and displayed in the user's local time.

Generate the identifier once per logical submission and reuse it for retries. Enforce uniqueness in storage, reconcile ambiguous responses and reject or explicitly reconcile conflicting payloads with the same key. Track generated text, model or prompt version, and human review state separately from approved learning material. AI output can be linguistically incorrect, so review is part of the workflow.

Entity	Key fields	Purpose
Words	WordID, source text, target text, language	Canonical vocabulary
Examples	ExampleID, WordID, level, text, reviewed	Generated or curated examples
Learners	LearnerID, preferences, time zone	Personalization
Attempts	AttemptID, WordID, LearnerID, timestamp, correct	Event history
Schedule	LearnerID, WordID, due time	Practice selection

Separate canonical answers from generated examples. Store the actual submitted answer and assessment rule version, subject to privacy requirements. A durable attempt table makes it possible to recompute metrics when assessment policy changes. Avoid storing unnecessary personal information, and do not transmit sensitive content to an unapproved AI service.

9.2 Microsoft-oriented architecture



Responsibility	Microsoft implementation	Conventional implementation
Canonical storage	Dataverse or PostgreSQL connector	PostgreSQL tables and constraints
AI enrichment	Approved Copilot/AI service and flow	Python client and worker
Quiz UI	Power Apps with Power Fx	Shiny, Streamlit or web frontend
Scheduling	Power Automate or managed scheduler	Cron, queue or scheduler service

Responsibility	Microsoft implementation	Conventional implementation
Preparation	Power Query (M)	PostgreSQL views, Python or R pipeline
Interactive analytics	Power BI measures in DAX	SQL metrics and reactive dashboard
Retention modeling	Validated R/Python analysis	R/Python analysis

The conventional stack is not automatically simpler: authentication, hosting, notifications and accessibility require engineering. The Microsoft stack can reduce those costs when the institution already has managed infrastructure, but it creates connector and licensing dependencies. Start with a small, approved repository and a simple quiz. Add scheduling and analytics after the data contract and review process are stable. There is no benefit in introducing M if all metrics are already prepared in an authoritative source and no additional shaping is needed.

9.2.1 Trace a single attempt through the system

A learner submits a translation in the app. Power Fx checks the input and requests a durable write through an approved connector. The storage service validates keys and permissions and returns success or failure. A scheduled process can later select due words; it is separate from the app's immediate response. A background AI enrichment process should write proposed examples with provenance and review status, not replace approved translations without validation.

Power Query can prepare a read-only analytical dataset from the attempt history, learner and word dimensions. For an Import model, that dataset becomes visible after a successful refresh; a learner's new attempt does not instantly appear in an existing cached report. When a report consumer selects a language or date, DAX recomputes accuracy under those filters. If a statistical model in R estimates retention as a function of practice interval, publish validated model outputs at a declared grain; do not confuse those estimates with observed quiz accuracy. A conventional PostgreSQL/Python application can implement the same flow with a transactional API and scheduled worker, while a Shiny report can expose interactive analysis. The important boundary is between durable writes, repeatable preparation, contextual descriptive metrics and statistical inference.

9.3 Illustrative measures and analytical decisions

Assume an Attempts model with one row per completed quiz response and a Boolean Correct field. A DAX measure for accuracy is:

DAX

```
Attempt Count =
// Count completed responses at the attempt grain
COUNTROWS (Attempts)
```

DAX

```
Correct Count =
// Intersect correctness with existing filters and count zero matches
COALESCE (
    CALCULATE ([Attempt Count], KEEPFILTERS (Attempts[Correct] = TRUE ())),
    0
)
```

DAX

```
Accuracy =
// Divide contextual correct attempts by contextual attempt count
DIVIDE ([Correct Count], [Attempt Count])
```

For eight correct and two incorrect attempts, accuracy is 80% overall, 100% under a Correct=TRUE filter and 0% under Correct=FALSE. An empty attempt population yields blank accuracy. If the business definition ignores correctness filters, remove those filters from both measures instead.

PostgreSQL can compute the corresponding unfiltered measure using `avg(Correct::integer)`, R can use `mean(Attempts$Correct)` and pandas can use `Attempts.Correct.mean()`, assuming nonmissing Boolean values. These are equivalent for the specified grain and complete data. DAX evaluates accuracy under report filters; in SQL or Python a query or reactive callback must supply equivalent filters. Accuracy alone is not a valid estimate of long-term retention: difficult words may be selected more often, practice intervals vary, and missing responses are not random. A statistical evaluation should define the target estimand and account for selection and repeated measurements.

A useful first experiment is to compare response accuracy after different practice intervals. Randomized assignment, if feasible, improves causal interpretation. R or Python can fit a model with learner and word effects; DAX is appropriate for descriptive monitoring of the resulting estimates. The analysis is distinct from the scheduling algorithm and should be validated before it informs an adaptive policy.

Takeaway. This example illustrates complementary layers without requiring every technology. The right architecture depends on data ownership, security, integration requirements, expected scale, and the team's support capacity.

10 Technical learning, adoption and mastery

10.1 Mapping transferable expertise

Existing expertise in programming, data engineering and public-sector work can bridge technical teams and business users. Teaching sound principles through familiar tools requires recognizing where Microsoft implementations differ from conventional development. The following material is a general learning and training framework.

Existing strength	Transferable concept	New learning priority
PostgreSQL	Joins, aggregation, schema, optimization	Delegation, folding and semantic relationships
R and dplyr	Functional data manipulation, reproducible analysis	M types, row lambdas and connector behavior
Python	Automation, API interaction and validation	Power Platform runtime and governance boundaries
Statistical modeling	Grain, weighting, uncertainty, validation	DAX filter context and non-additive measures
Training	Scaffolding, feedback and practice	Role-specific digital adoption and support

The six languages are not interchangeable. A reliable mental model identifies which operations transfer directly and which change when implemented in a managed ecosystem. Identify gaps in understanding explicitly and validate them through experiments rather than assuming that familiarity with one runtime establishes proficiency in another.

10.2 Principles for technical learning and adoption

Transferable foundations. SQL, R and Python provide a foundation for understanding filtering, joins, type conversion and aggregation. Focus additional study on the way the Microsoft environment executes those operations, particularly delegation, query folding and DAX filter context.

Teach the mental model. Start with a business problem rather than a list of features. Build a small app that collects information, prepare that information with Power Query and produce a Power BI report. Each stage introduces a different set of concepts in a recognizable context.

Different levels of training. Not every user needs to write DAX or M. Some need to interpret a dashboard accurately, some need to build reports, and others need to maintain semantic models. Tailor learning outcomes to those responsibilities and create opportunities for deeper technical skills.

Responsible citizen development. Low-code tools can broaden participation in software development, but ease of construction does not eliminate quality assurance. Teach validation, appropriate data access, testing and documentation alongside the tools themselves, and establish clear thresholds for involving professional engineering teams.

Evaluate adoption, not attendance. Determine whether users can complete a task independently after training, whether the solution is reliable, and whether it saves time without sacrificing quality. Attendance and satisfaction are useful signals, but do not prove sustained adoption.

Respect existing architecture. Do not migrate a reliable database transformation into Power Query merely because a solution uses Power BI. First understand where data and metrics are governed, then select the most maintainable execution layer and avoid duplicated work.

10.3 Core questions and answers

10.3.1 How do Power Query and DAX differ?

Power Query prepares and integrates data before or as it is retrieved into a semantic model, whereas DAX defines analytical calculations over that model. A static data-cleaning operation belongs upstream; a measure that responds to slicers belongs in DAX. There are overlaps, but the difference in execution timing and context is fundamental.

10.3.2 Why might a Power Apps formula work in a demo and fail at scale?

The connector may not delegate an expression to the source. With a small collection the app can evaluate the full dataset, but a nondelegable query against a large remote source may process only an initial subset. Inspect delegation warnings, choose supported predicates and test using a dataset larger than the local limit.

10.3.3 Why is a Power BI total sometimes different from the sum of visible rows?

A measure is evaluated in the total cell's broader filter context. That is necessary for non-additive metrics, such as distinct customer counts and profit margins. A percentage should generally be recalculated from the underlying numerator and denominator rather than added or averaged across categories.

10.3.4 How would you explain query folding to a nontechnical employee?

Imagine asking a database to send you only the relevant records rather than downloading everything and deleting unwanted rows on your computer. Query folding allows Power Query to pass supported steps back to the data source. This can reduce data transfer and refresh time.

10.3.5 How would you decide between low-code and professional development?

Consider sensitivity, transaction integrity, complexity, expected use, support requirements and integration. A small internal form might be ideal for Power Apps; a critical multi-system workflow may need a dedicated engineering team and stronger lifecycle controls. Developers should follow the organization's established policies and escalation procedures.

10.3.6 How would you approach AI adoption in this ecosystem?

Pair practical use cases with validation and responsible use. For example, AI can draft a Power Query transformation, but employees should understand how to inspect the resulting steps, verify output against fixtures and protect sensitive information. Training should develop judgment, not just prompt-writing habits.

10.4 Extended questions and answers

10.4.1 Foundations and execution

Why do specialized languages coexist with SQL, R and Python? They offer different execution and integration contracts. A reactive formula controls an application property; a repeatable transformation prepares data; a semantic measure evaluates a business definition under analytical filters. General-purpose languages can implement these capabilities, but usually require application frameworks or additional infrastructure.

Is a functional language necessarily lazy or immutable? No. Functional composition concerns how expressions and functions are combined. Laziness concerns when they are evaluated, and immutability concerns whether a value

changes. M combines these features in specific ways; Python supports first-class functions but also mutation and mostly eager evaluation.

What does a local binding buy me? It makes intermediate dependencies explicit and can simplify testing. M's `let` bindings describe values, DAX `VAR` captures a result in its defining context, and Power Fx `With()` introduces names for an expression. They resemble familiar variables but do not necessarily behave like reassignment in Python.

Why can missing values produce different aggregate results? SQL aggregates commonly ignore nulls; R's `mean()` defaults to propagating missingness; pandas numeric reductions commonly skip missing values; DAX can return blank for an empty context. Decide what missingness means before comparing outputs.

10.4.2 Power Fx and M

Why would a label show a stale value? If it references a stored variable set during an earlier event, changing an input does not automatically repeat that assignment. Use a property or named formula for derived state and reserve mutable variables for state that genuinely needs independent persistence.

Does `Patch()` provide the same guarantee as a PostgreSQL transaction? No. It requests a connector operation; a successful individual write does not imply atomicity across multiple requests. Use server-side transactions or an appropriately designed service for operations requiring all-or-nothing behavior.

How can a nested lookup silently return the wrong product? If the inner and outer records have identical field names, an ambiguous predicate may resolve both references to the inner scope. Use aliases with `As` to explicitly distinguish the current order and the candidate product.

What happens when folding fails? In a conventional Import query, the engine may retrieve more rows or columns and evaluate unsupported transformations locally. This usually changes cost rather than the intended result. DirectQuery imposes additional constraints. A nondelegable Power Fx operation can instead return an incomplete answer because it only evaluates a bounded local subset.

Does `each` introduce a loop? It introduces an anonymous one-argument function. `Table.SelectRows()` invokes that predicate for candidate records, or translates it if supported. The syntax specifies the predicate, not a guaranteed physical row-by-row execution plan.

Should I buffer every slow query? No. Buffering can materialize values and inhibit further folding, increasing memory and network work. Measure the actual bottleneck, inspect source requests, and use buffering only for a demonstrated reason.

When should PostgreSQL own the transformation? When relational logic is authoritative, reused widely, sensitive, expensive or dependent on database constraints and indexes, a tested source view or pipeline is often preferable. M is useful for consuming heterogeneous sources and report-specific preparation without modifying the source system.

10.4.3 Power BI architecture and modeling

What is published when I publish a Desktop report? Depending on the project and connection, publishing creates or updates report and semantic model artifacts in the service. A report using an existing shared model can be separate from that model. Access, refresh and source connectivity require their own configuration.

Can I change a slicer without refreshing the model? Yes. An Import model visual normally submits a new analytical query over the already loaded data. Refresh changes stored source data; interaction changes the evaluation context. DirectQuery may send a new source request, but the concepts still differ.

Why is a semantic model not merely a database? It includes reusable analytical definitions, relationships, formatting, hierarchies, security and storage configuration. A database supplies data and integrity mechanisms, while the model determines the analytical meaning of filters and measures. It can depend on a database without copying all of its tables.

Why is a star schema helpful? A clear fact grain and unique dimensions allow predictable filtering. Selecting a product restricts the related facts without needing every visual to reconstruct joins. Ambiguous many-to-many paths and mixed grains can create incorrect totals even when individual formulas look reasonable.

A product slicer does not change a revenue card. What should I inspect? Check whether the slicer uses the intended dimension, whether the relationship is active and directed towards `Sales`, whether the visual interaction is enabled, and whether the measure clears the relevant filters. Compare a basic sum with the suspect measure before changing the model.

When is DirectQuery preferable to Import? Consider requirements for freshness, source access, model size, data residency and capacity. DirectQuery transfers interactive query workload to the source and can increase latency. Import trades refresh management and storage for interactive performance. Measure both under the actual workload rather than choosing based on a slogan.

What does a gateway do? It enables supported cloud-to-source connectivity where the service cannot reach a source directly. It is not a replacement for permissions or the analytical engine. Check connectivity, credentials and refresh or query mode when diagnosing a failure.

10.4.4 DAX contexts and analytical reasoning

Why does DAX need a filter context if I already have row context? Row context gives an expression access to current-row values. Filter context specifies the permitted data for an aggregation. An iterator can read a product key without automatically restricting a naked sum of fact rows. Context transition connects these two mechanisms when needed.

What changes when I wrap a sum in `CALCULATE()` in a `Products` calculated column? The current product row becomes a filter, which propagates through the active relationship. The original naked sum may return 2,950 for each product; the transitioned sum returns 1,300, 450 or 1,200 for the corresponding product.

Why can a measure reference inside `AVERAGEX()` behave differently from an inline `SUM()`? Evaluating a measure reference within row context triggers implicit context transition. A naked sum does not. For the fixture, averaging customer-specific revenue yields 983.33 rather than repeating the grand total.

What is the difference between `FILTER()` and `CALCULATE()`? The former produces a table of qualifying rows; the latter evaluates a scalar in a modified context. A table returned by `FILTER()` can be passed into `CALCULATE()` to express a complex restriction.

Why does `CALCULATE()` sometimes override a slicer? A simple Boolean filter on a column normally replaces the existing filter on that column. Use `KEEPFILTERS()` to intersect instead. Other filters can remain and row-level security is not bypassed.

Which denominator should a product share use? Define the business question. All products, selected products and products in the current category produce different denominators. Use targeted filter removal or an outer-selection calculation and test detail cells and totals.

What distinguishes `ISINSCOPE()` from `HASONEVALUE()`? Grouping level and visible-value count are different. A slicer can restrict a column to one value even when the visual does not group by that column. A hierarchy-aware total should generally inspect the grouping level.

Why can adding monthly distinct customer counts overstate annual customers? A customer may purchase in multiple months. An annual distinct count must deduplicate over the whole year. The same principle explains why ratio totals must be recalculated using the appropriate numerator and denominator.

Why use a separate `Calendar` dimension? It represents periods even when no fact rows exist, provides consistent grouping and supports time shifts across a contiguous date range. A date extracted only from observed transactions cannot represent missing days reliably.

Why might a variable ignore a later context change? It holds the value evaluated where it is defined. Put the measure evaluation inside `CALCULATE()` before binding the result if the variable must represent a modified context.

10.4.5 Troubleshooting, design and governance

A report is slow. Should I replace every iterator? No. Use Performance Analyzer to identify expensive visuals and inspect generated queries. DAX Studio can help separate Formula Engine and Storage Engine timings. Investigate source latency in DirectQuery, high cardinality and excessive visual grouping. Change the bottleneck you can measure.

What is the difference between a transformation test and a semantic model test? A transformation test verifies rows, types, keys and values after preparation. A model test verifies measures under filters, relationships, security and totals. Correct data preparation does not guarantee correct analytical semantics.

When should a calculation stay in M rather than DAX? If it is a stable row-level attribute needed for storage or grouping, calculate it upstream where practical. If the result must respond to arbitrary report selections, a DAX measure is generally more suitable. Consider refresh cost, storage and reuse.

When is Shiny or a Python framework preferable to a canvas app? When custom interaction, portability, complex transactions, specialized computation or integration requirements exceed what the managed host provides. Conversely, a bounded form with approved connectors can be much easier to maintain in Power Apps. Compare lifecycle costs and governance, not only initial construction effort.

How should AI-generated formulas be validated? Treat generated code as an untrusted draft. Inspect the semantics, run small known fixtures, test error conditions, verify data access and review performance. Do not send restricted data to unapproved systems or accept a plausible-looking chart as proof of correctness.

How can I tell whether I understand these systems? Reproduce the fixture results, predict what happens after changing a relationship or filter, explain which engine executes a transformation, and deliberately introduce a faulty example before repairing it. Being able to diagnose a surprising result is stronger evidence of mastery than remembering function names.

10.5 Training demonstration design

Audience	Exercise	Evidence of learning
General staff	Filter a dashboard and interpret margin	Explain why a total is not a sum of percentages
Report creators	Merge two files, validate keys, refresh	Reconcile row counts and detect duplicates
Citizen developers	Build a validated quiz or intake form	Handle missing inputs and failed writes
Advanced creators	Diagnose delegation and filter context	Explain and correct an incorrect result
Team leads	Assess low-code versus custom solution	Identify risks, ownership and escalation

A practical workshop should begin with a short demonstration, then give participants an independent task and an intentionally flawed example to diagnose. Provide a reference sheet and office hours for later practice. Evaluate time-to-completion, accuracy, reuse, support requests and user confidence over a defined interval, with suitable privacy protections. Avoid claiming productivity gains without measuring comparable tasks.

10.6 Misconceptions to address

“Power BI is just a charting tool.” In fact, Power BI combines integration, semantic modeling and analytics.

“Power Query replaces SQL.” Power Query complements SQL, especially across heterogeneous sources.

“Low-code eliminates engineering.” Low-code shifts the implementation interface, not quality obligations.

“DAX is just Excel formulas.” The distinctive behavior of DAX comes from evaluation context.

“Copilot-generated code is reliable by default.” Generated code requires review, tests and data governance.

"More training sessions mean better adoption." Adoption depends on demonstrated use and sustained outcomes, not simply the number of sessions.

Explain these distinctions in plain language before introducing technical details. Adapt explanations to the learner's questions rather than reciting terminology. The aim is independent problem solving, collaboration with specialists, and sound architectural decisions.

Takeaway. Combine conventional programming knowledge with the organizational realities of digital adoption. Use specific examples to translate technical expertise into measurable learning outcomes.

11 Reference sheets

11.1 Function families

Task	Power Fx	M	DAX
Filter	`Filter()`	`Table.SelectRows()`	`FILTER()`
Projection	`ShowColumns`	`Table.SelectColumns()`	`SELECTCOLUMNS()`
Extend	`AddColumns()`	`Table.AddColumn()`	`ADDCOLUMNS()`
Group	`GroupBy()`	`Table.Group`	`SUMMARIZECOLUMNS()`
Aggregate expression	Sum	`List.Sum()` + transform	`SUMX()`
Unique values	`Distinct()`	`Table.Distinct()`	`DISTINCT()` / `VALUES()`
Local binding	`With()`	`let` / `in`	`VAR` / `RETURN`
Missing fallback	`Coalesce()`	`??`	`COALESCE()`
Relationship lookup	`LookUp()`	`Table.NestedJoin()`	`RELATED()`
Date arithmetic	`DateAdd()`	`Date.AddDays()`	`DATE()` / date arithmetic
Error recovery	`IfError()`	`try` / `otherwise`	`IFERROR()` / `DIVIDE()`
Data write	`Patch()`	No general equivalent	No general equivalent

Treat this as a navigation index, not a list of interchangeable functions. Table functions, scalar measures and behavior functions have different output shapes and side effects. In particular, DAX `VALUES()` may represent an unknown dimension member; Power Fx `Distinct()` is a single-column table; M `Table.Distinct()` deduplicates according to specified criteria. Use the worked drills for exact patterns.

Task	PostgreSQL	R	Python
Filter	`WHERE`	`filter()`	`query()` / `loc[]` column indexing
Projection	`SELECT`	`select()`	
Extend	`SELECT ... AS`	`mutate()`	`assign()`
Group	`GROUP BY`	`group_by()` + `summarise()`	`groupby()` + `agg()`
Join	`JOIN`	`left_join()`	`merge()`
Distinct	`DISTINCT`	`distinct()`	`drop_duplicates()`
Window	`OVER`	dplyr window functions	`rolling()` / `rank()` / `transform()`
Nesting	`jsonb` / arrays	list columns	`dict` / `list`
Missing fallback	`COALESCE()`	`coalesce()`	`fillna()`
Error recovery	PL/pgSQL exception blocks	`tryCatch()`	`try` / `except`

R and Python examples depend on packages. SQL aggregation uses relational semantics and may ignore `NULL` values. R's `mean()` returns `NA` if an `NA` is present unless instructed otherwise; pandas generally skips missing numeric values. Align policy before comparing metrics.

11.2 Glossary

Term	Meaning
Grain	What one row of a fact table represents
Semantic model	Business-oriented tables, relationships and metrics
Row context	Current row used for expression evaluation
Filter context	Constraints governing an analytical calculation
Context transition	Conversion of row context into filter context
Delegation	Remote execution of supported Power Fx operations
Query folding	Translation of M operations into source-native requests
Lineage	Connection between virtual DAX columns and model origins
Cardinality	Number of distinct values or relationship multiplicity
Reactive dependency	Recalculation when an input dependency changes
Idempotency	Repeating an operation does not duplicate its effect
RLS / OLS	Row-level / object-level security
ALM	Application lifecycle management
VertiPaq	Columnar in-memory analytical storage engine

The vocabulary connects existing software-engineering and data-science knowledge to the Microsoft-specific execution model. If a term is unfamiliar, revisit the relevant chapter and reproduce the corresponding worked example. In particular, do not confuse row context with a SQL `WHERE` clause or delegation with guaranteed query folding.

12 Sources and further reading

The numbered references below support the documented language and platform concepts. The decision matrices and case study are technical recommendations, not statements of vendor or organizational policy. The code was reviewed for intended semantics and fixture results were independently checked with a Python script; the Microsoft runtimes, R and PostgreSQL were not available for executing these examples in this environment. Host-specific syntax and availability should be verified before deployment.

1. Microsoft Learn. Power Fx overview; Canvas apps formula reference; Behavior formulas.
2. Microsoft Learn. Delegation in canvas apps; Power Fx tables; App object and named formulas.
3. Microsoft Learn. M language specification; M function reference.
4. Microsoft Learn. Query folding; Privacy levels; Table.Buffer.
5. Microsoft Learn. DAX reference; CALCULATE; Filter functions; Avoid FILTER as a filter argument.
6. Microsoft Learn. DAX basics; Time intelligence; Calculation options.
7. Microsoft Learn. Storage modes; Star schema guidance; Query folding guidance.
8. SQLBI. Understanding context transition; Formula and storage engines; Technical articles and DAX training; DAX Guide; DAX Patterns.
9. PostgreSQL documentation. Table expressions; Window functions; Query plans.
10. R Project. R language definition; R manuals; tidyverse dplyr, tidyr, purrr, stringr, lubridate, dbplyr.
11. Python Software Foundation. Python documentation; pandas user guide; GroupBy; Merging; NumPy documentation; Requests.
12. Microsoft Learn. Power Platform ALM; Power BI Desktop projects. These are starting points for checking current lifecycle and source-control support.
13. Microsoft Learn. Query plans; Native query folding; PostgreSQL connector; Folding indicators.
14. Microsoft Learn. End-to-end Power BI tutorial; Semantic models.
15. Microsoft Learn. Table storage modes; Semantic model modes; Direct Lake overview.
16. Microsoft Learn. DAX overview; CALCULATE; KEEPFILTERS.
17. Microsoft Learn. Power Query terminology; Power Apps code readability; Power Query formatter.
18. SQLBI. DAX style; DAX formatting.
19. Tidyverse style guide; PEP 8; PEP 257; SQLFluff rules.
20. Microsoft Learn. Power Fx data types.
21. Microsoft Learn. Data types in Power Query.
22. Microsoft Learn. Data types in Power BI Desktop.
23. Microsoft Learn. M language basic concepts.
24. Microsoft Learn. Use DAX user-defined functions.
25. Microsoft Learn. Error, IfError, IsError, and IsBlankOrError functions.
26. Microsoft Learn. M language error handling.
27. Microsoft Learn. Behind the scenes of the Data Privacy Firewall.
28. Microsoft Learn. Roles in workspaces in Power BI.
29. Microsoft Learn. Visual calculations overview.
30. SQLBI. Context transition and filters in CALCULATE.

31. SQLBI. Context transition and expanded tables.
32. SQLBI. Understanding data lineage in DAX.
33. Microsoft Learn. RELATED.
34. Microsoft Learn. ISINSCOPE.
35. PostgreSQL 16. Aggregate functions.