Tabular Editor

Optimizing Semantic Model Size and Refresh in Fabric Capacities

Power BI Gebruikersdagen 2026

Meet the speakers

Greg Baldini
Core Developer

Just Blindbæk
Field CTO

A BIG thank you to the partners

Tabular Editor

Agenda

1. Foundations

   Fabric memory limits and the VertiPaq engine

2. Model Size Optimization

   Core techniques and custom aggregations

3. Refresh Optimization

   Partition strategies, Incremental refresh, custom partitions, hybrid tables, Direct Lake & Refresh scale-out.

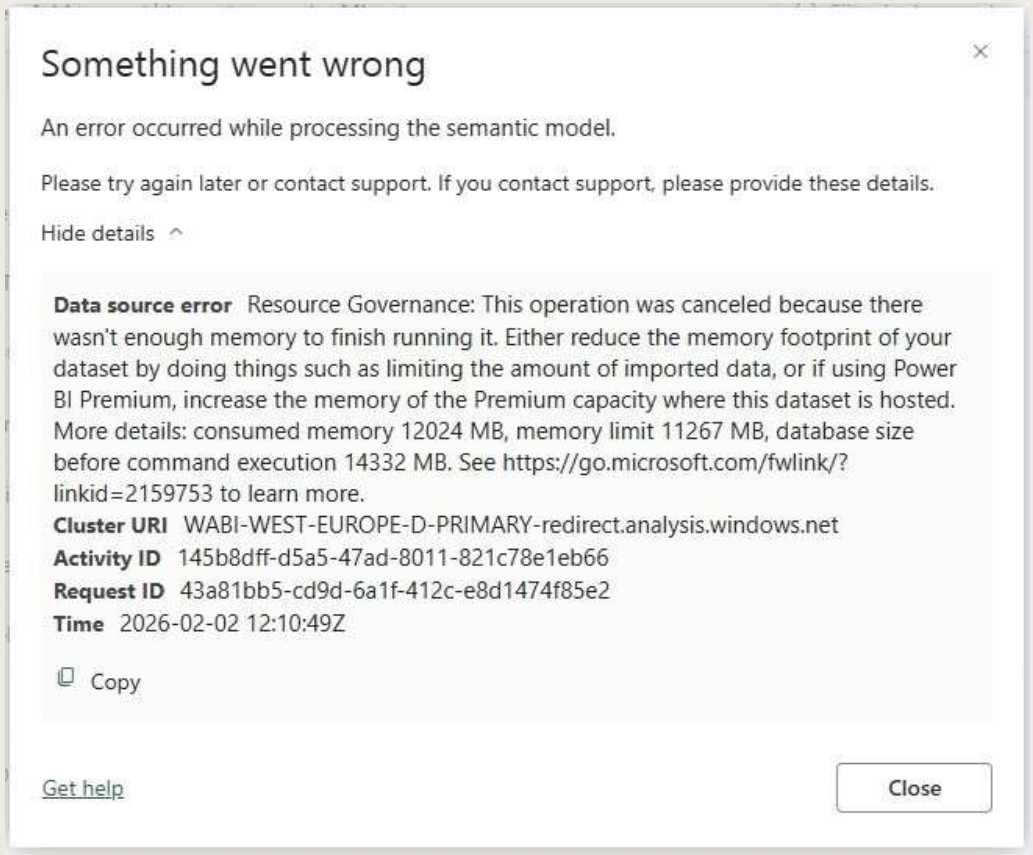
Foundations

Fabric Capacity Limits

Fabric Capacity Limits

1. Fabric Capacities Are Powerful – Memory Is a Bottleneck
2. Fabric SKU Memory Limits (Model Size)
3. Model Size ≠ Refresh Memory Usage
4. The Design Choice - You Always Pay Somewhere
5. Where Does the Memory Go?

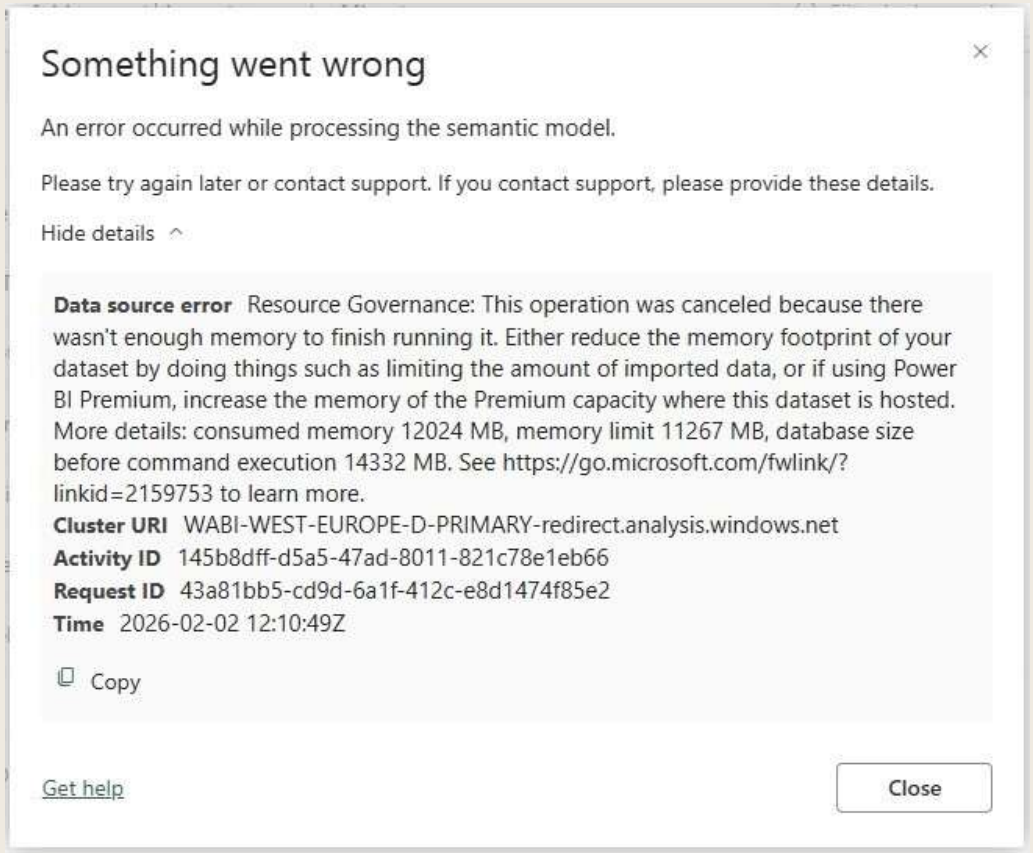
Fabric Capacities Are Powerful – Memory Is a Bottleneck



Fabric Capacities Are Powerful – Memory Is a Bottleneck

- Optimizing model memory is nothing new. It's an ongoing challenge since the VertiPaq engine first appeared as PowerPivot in Excel 2010
- It's easy (and common) to hit your memory limits
- Refresh requires additional peak memory

- *You don't run out of features. You run out of memory.*



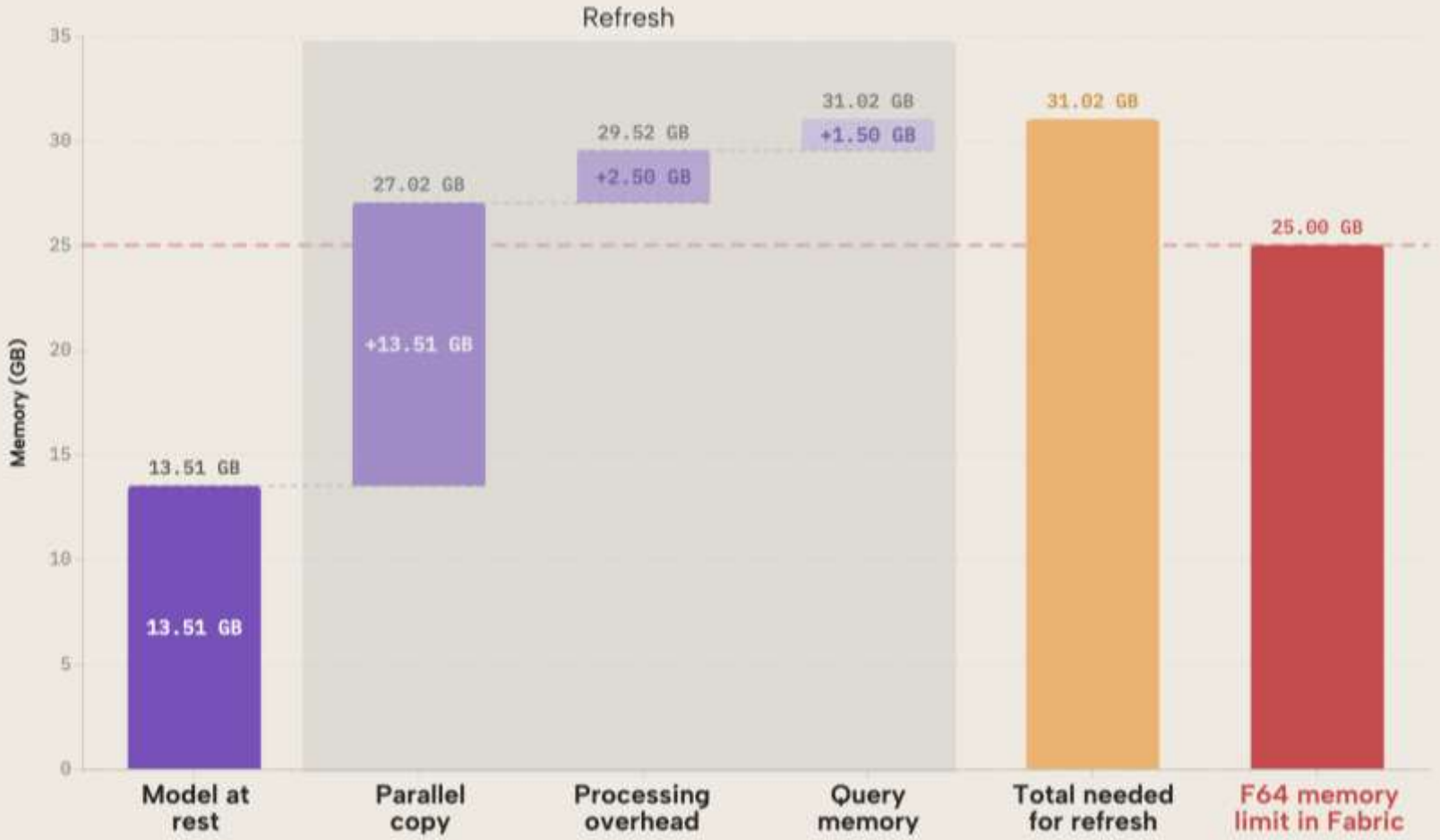
Tabular Editor

Fabric SKU Memory Limits (Model Size)

- There are semantic model memory limits in Power BI Pro, PPU, and Fabric
- Memory is a hard ceiling; it's not flexible or adaptive like compute
- SKU memory limits are enforced per semantic model
- Capacity upgrades are not linear

SKU	Max Model Size
Pro	1 GB
F2	3 GB
F4	3 GB
F8	3 GB
F16	5 GB
F32	10 GB
F64	25 GB
F128	50 GB
PPU	100 GB
F265	100 GB

Steady-State vs Peak Memory



Tabular Editor

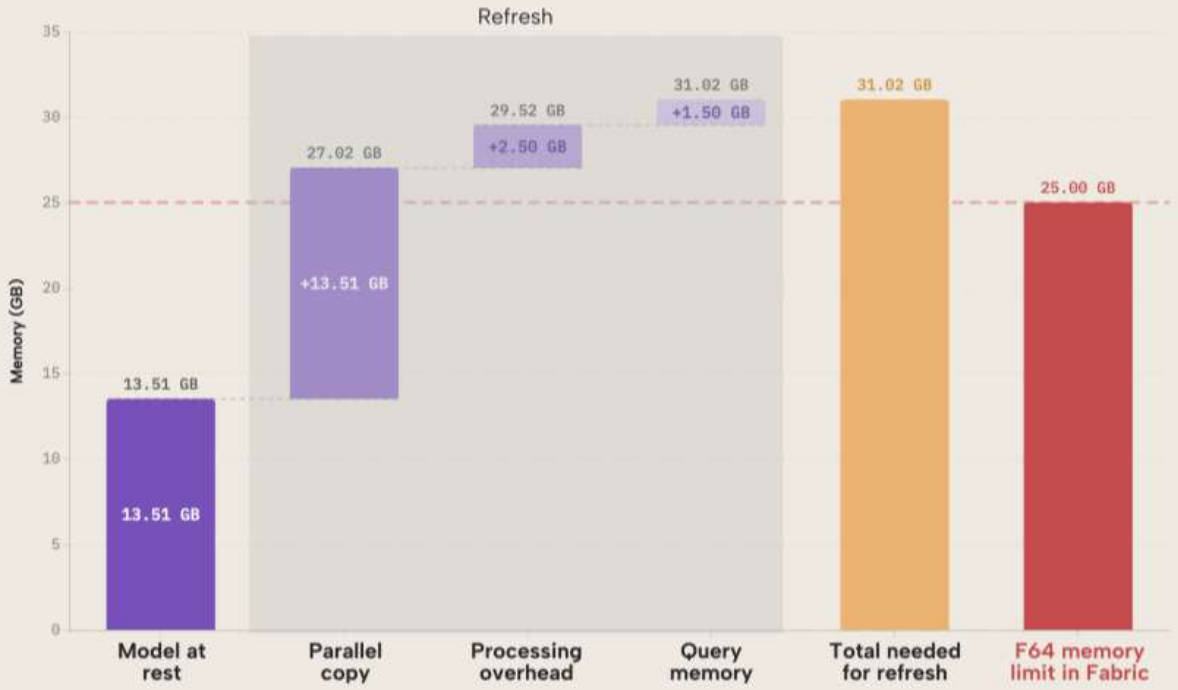
Steady-State vs Peak Memory

Steady-State Memory

- Model in memory
- User queries

Peak Memory (Refresh)

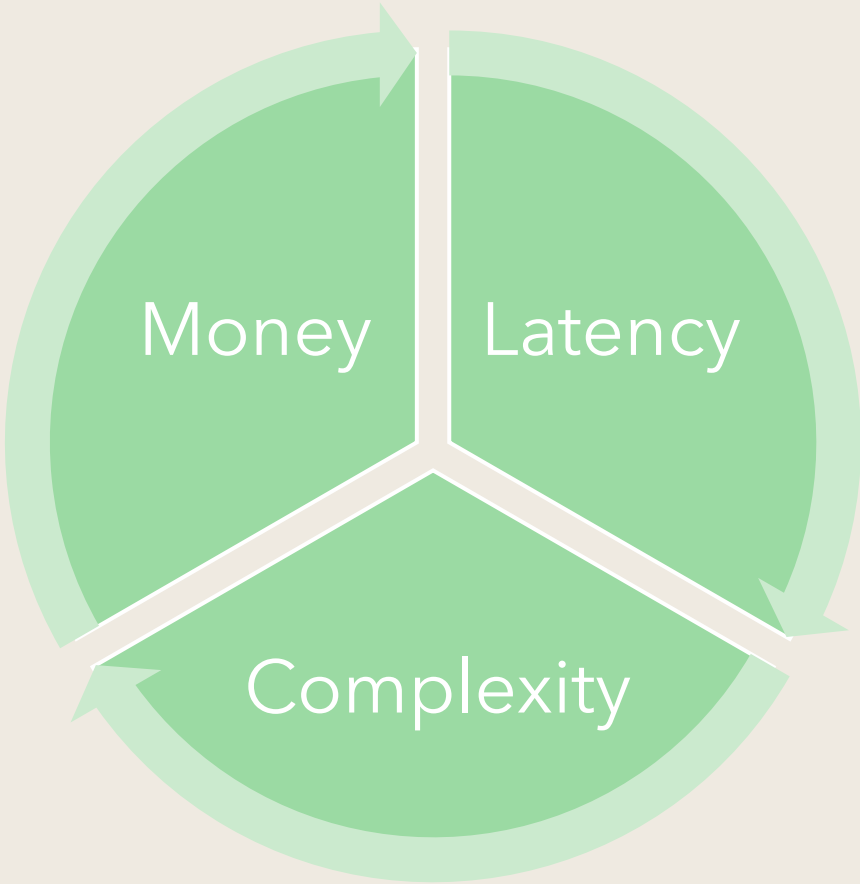
- New segments built
- Old segments still present
- Memory temporarily spikes



Tabular Editor

The Design Choice – You Always Pay Somewhere

- Upgrade SKU instead of redesigning the model
- Move full model or details to DirectQuery
- Introduce partitions, hybrid tables, or scale-out



- *Optimization is not about eliminating cost – it's about choosing where to pay it*

Tabular Editor

Where Does the Memory Go?

Dictionaries

Hash tables mapping unique values to integer IDs for efficient storage

Data Segments

Compressed column partitions storing encoded row values

Attribute Hierarchies

Pre-built aggregation structures for natural hierarchy navigation

Metadata

Schema definitions, relationships, and model configuration info

VertiPaq Internals

VertiPaq Internals

1. Dictionaries
2. Segments
3. Hierarchies
4. Compression (RLE hint)
5. Processing memory vs steady-state memory

Data storage format

Some data

2	-2832.21	0	Lorem
8	33289.8	1	ipsum
7	43987	1	dolor
5	92348.1	1	sit
1	7	0	amet
3	232.832	1	consectetur
4	0.00984	1	adipiscing
6	2048	0	elit,

This data has

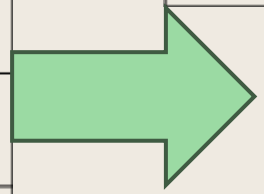
- 4 columns
- 3 types
- 8 rows

- How do we store it?

Data storage format
Row-wise

Some data

2	-2832.21	0	Lorem
8	33289.8	1	ipsum
7	43987	1	dolor
5	92348.1	1	sit
1	7	0	amet
3	232.832	1	consectetur
4	0.00984	1	adipiscing
6	2048	0	elit,



As rows

2	-2832.21	0	Lorem

8	33289.8	1	ipsum

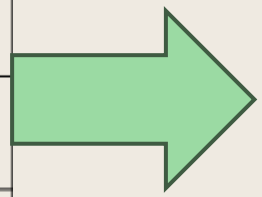
4	0.00984	1	adipiscing

6	2048	0	elit,

Data storage format
Column-wise

Some data

2	-2832.21	0	Lorem
8	33289.8	1	ipsum
7	43987	1	dolor
5	92348.1	1	sit
1	7	0	amet
3	232.832	1	consectetur
4	0.00984	1	adipiscing
6	2048	0	elit,



As columns

2
8
7
5
1
3
4
6

-2832.21
33289.8
43987
92348.1
7
232.832
0.00984
2048

0
1
1
1
0
1
1
0

Lorem
ipsum
dolor
sit
amet
consectetur
adipiscing
elit,

VertiPaq is a compressed columnstore engine

Compression

- Value encoding
- Dictionary / hash
- Run-length

VertiPaq compression trades up-front processing time for smaller size in RAM

All compression algorithms are optimized for fast decompression (very cache friendly)

VertiPaq types

Vertipaq Type
64-bit integer
64-bit float
Bit
String

VertiPaq types

Vertipaq Type
64-bit integer
64-bit float
Bit
String

Surprised?

VertiPaq types

Vertipaq Type	DAX / Tabular type
64-bit integer	• Whole number • Fixed decimal (currency): divide by 10,000
64-bit float	• Decimal • Date/time: whole portion as days since 1899-12-30, decimal portion as fraction of day since 00:00:00 • Date: whole portion as days since 1899-12-30 • Time: [0, 1) as fraction of day since 00:00:00
Bit	True/false
String	Text

Value encoding

Raw column

Number
17,000,000,001
17,000,000,002
17,000,000,017
17,000,000,028
…
17,000,000,031

- What do these have in common?
- What varies?
- How big is the portion that varies?

(assume all numbers bounded in range shown)

Varying part is only 0-31: representable in 5 bits

By storing only 5-bit offset, we save >90% of space

Value encoding

- Integers only
- As simple as base + offset
- VertiPaq uses more sophisticated mathematical relationships among values as well
  - These are not specifically documented publicly, so far as I know

Dictionary encoding

Raw

String
A long string
Lorem ipsum
Lorem ipsum
999 characters
A short string
999 characters

Dictionary

String	Dict Key
A long string	00
Lorem ipsum	01
999 characters	10
A short string	11

Encoded

String encoded
00
01
01
10
11
10

Dictionary encoding

- Works on any data type
- Minimal-bit keys to encode distinct values
- Compression ratio inversely proportional to cardinality
  - Useless for PK fields or other unique fields
- AKA "Hash"

Run-length encoding

Raw

Date
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-01-01

Run-length

Date	Count
2023-01-01	6
2023-02-01	1

Run-length encoding

- Works on any data type
- Depends entirely on sort order of column
- Compression ratio proportional to the number and length of repeated value runs
  - Useless for PK fields or other unique fields
- VertiPaq uses heuristics and tries multiple sort orders for each table
  - Table has one sort order, so the optimal sort is very difficult to predict
  - **Does** take into account the sort order of the incoming rows
    - **May** choose a different sort order; table sort order is not configurable, only influence-able

Under (RAM, parallelism) pressure

RAM hogs

- Value encoding and dictionary encoding both require holding in RAM all unique values being compressed. This is very expensive.
- Sort order sampling requires a huge amount of RAM.
- Querying a single, monolithic object, is not readily parallelizable.

So, VertiPaq has segments

- Subset of a column's values (fixed number of rows)
- Unit of processing (refresh)
- Unit of parallelism for queries
- Partitions have >=1 segments

Segments

Units of compression

- Dictionary bit keys per segment
  - Dictionary bit keys may point to different encoded values in different segments
- Each segment Is compressed independently

Units of processing

- First segment may be up to 2x DefaultSegmentRowCount
- Begin reading rows from source (for all columns)
- When we hit configured row count
  - Begin compressing segment
  - Begin reading rows for next segment
  - If first segment, split and begin compressing 2 segments
- Greater compression with larger segments, but also greater processing time

Units of parallelism

- VertiPaq queries (aka storage engine) are parallel based on the number segments
  - Query hitting one segment = 1 thread = no parallelism
- Because queries are distributed across cores, each segment's subquery yields a partial result
  - All partial results must be combined – single threaded, final task of a query
  - Too many segments create overhead
- Perfect query: 1 segment per CPU core

Tabular has partitions

- Again, VertiPaq has segments
- Partitions are a tool to subdivide a table's values, with a different query for each partition
- Partitions support things like incremental refresh (or any other reason to refresh only a subset of data)
- VertiPaq is ignorant of Tabular partitions, but segments cannot span partitions

- You can never really touch or control a segment. Partitions are what you define and manage

Recap

- Cardinality is primary determinant of compression ratio
- Larger segments offer better opportunities for high compression ratios
- Segments never span partitions
- Primary column size components
    - Dictionary
    - Data
    - Hierarchies

Model Size Optimization

Core Reduction Techniques

Core Reduction Techniques

1. Remove unnecessary data
2. Reduce cardinality
3. Disable unnecessary hierarchies

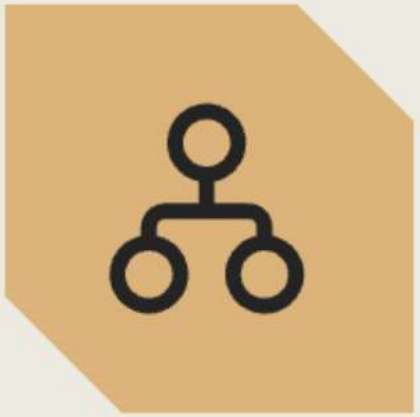
Remove unnecessary data

- The simplest and most effective way to lower semantic model size
- Only include the data required for analysis
- Audit fields, GUIDs, technical identifiers, and rarely used attributes frequently consume disproportionate memory

Reduce cardinality

- High-cardinality columns are one of the most common causes of large semantic models
- Transform the data to limit the number of unique values
- Remove unnecessary precision
- Avoid floating-point types (Double, Single, Float) for values that require exact precision, especially financial amounts. Use Fixed Decimal (Currency) where appropriate and Integers for identifiers and counts
- Splitting a DateTime column into separate Date and Time columns is often beneficial

Disable unnecessary hierarchies



- An attribute hierarchy is a piece of metadata that allows certain client tools (most notably Excel Pivot tables) to browse and interact with that column
- Disable it for surrogate keys, technical identifiers, GUIDs, and columns used only inside DAX measures
- The setting is called IsAvailableInMDX and defaults to true.

Strategic Detail Isolation

Strategic Detail Isolation

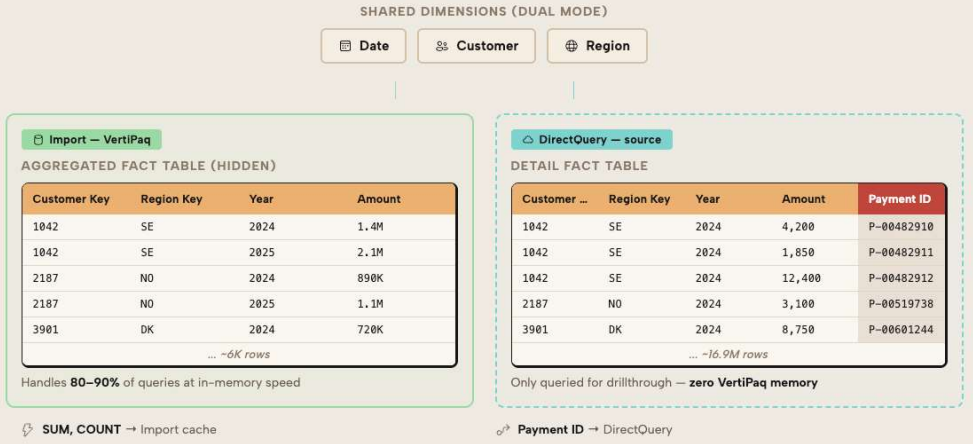
1. Remove historic details
2. Shadow table with aggregation mapping
3. Hybrid Storage Patterns

Remove historic details

- Removing high-cardinality columns is ALWAYS the first choice
- Cannot remove it? Then, only include data in the column for the last year?

User-defined aggregations

- Sometimes you cannot remove or reduce high-cardinality detail columns
- User-defined aggregations let you split a fact table into two logical layers. A smaller, aggregated Import table that handles most reporting queries, and a detailed DirectQuery table that retains the high-cardinality columns you can't afford to import
- The detailed DirectQuery table still exists, but it does not consume VertiPaq memory



User-defined aggregations

SHARED DIMENSIONS (DUAL MODE)

Date | Customer | Region

Import — VertiPaq

AGGREGATED FACT TABLE (HIDDEN)

Customer Key	Region Key	Year	Amount
1042	SE	2024	1.4M
1042	SE	2025	2.1M
2187	NO	2024	890K
2187	NO	2025	1.1M
3901	DK	2024	720K
... ~6K rows			

Handles **80–90%** of queries at in-memory speed

DirectQuery — source

DETAIL FACT TABLE

Customer ...	Region Key	Year	Amount	Payment ID
1042	SE	2024	4,200	P-00482910
1042	SE	2024	1,850	P-00482911
1042	SE	2024	12,400	P-00482912
2187	NO	2024	3,100	P-00519738
3901	DK	2024	8,750	P-00601244
... ~16.9M rows				

Only queried for drillthrough — **zero VertiPaq memory**

SUM, COUNT → Import cache

Payment ID → DirectQuery

Refresh Optimization

Processing Under Pressure

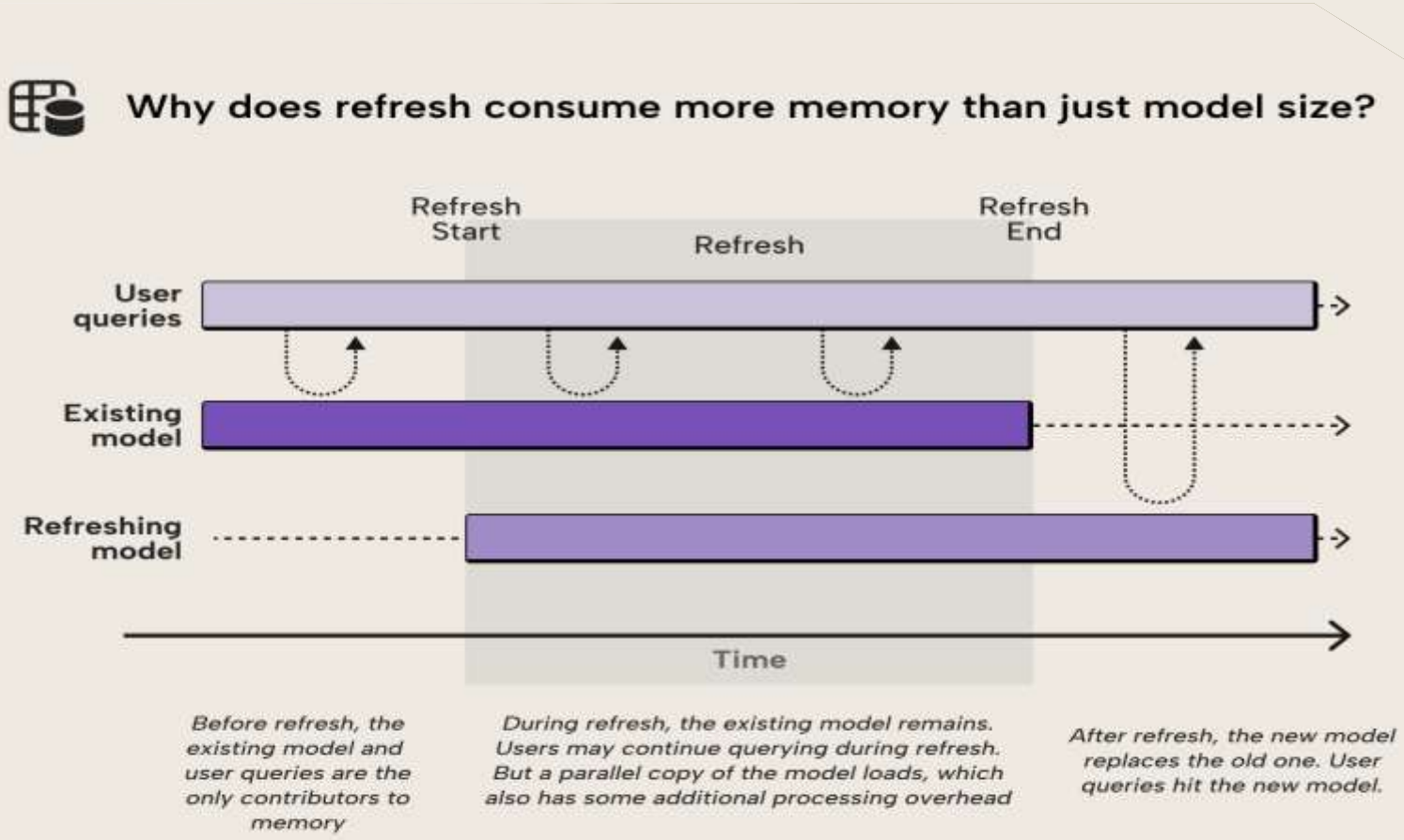
Processing Under Pressure

1. Why full refresh doubles memory
2. Partition strategies
3. Incremental refresh
4. Custom partitions
5. Hybrid tables
6. Direct Lake trade-offs

Refresh timeline and RAM

- Refreshing requires that we create new versions of all partitions being refreshed

- While partitions are processing, we still need to serve queries to users



Partitioning

- N partitions in a table
- Separate query per partition
- Separate storage mode per partition
- Partitions can be processed / refreshed individually
- Processing a partition requires extra RAM only in proportion to the size of the partition(s) being processed
    - RAM benefits accrue only with orchestration to refresh less than the whole model at once

Partition strategies

Incremental refresh

- Automatic in Power BI
- Policy-based on time ranges
- Automatically drops too-old partitions

Custom partitions

- Create partitions with a tool other than PBI Desktop (Tabular Editor 2 or 3, Visual Studio, editing TMSL/TMDL, XMLA commands to create partitions)
- Any query separation you would like
- Requires your own orchestration
- Gives much more control

Hybrid tables

- Built-in
  - Incremental refresh for old data
  - DirectQuery for the latest partition
- Custom
  - Any partition can be DQ and any import
  - E.g., DirectQuery for archival data, and import for latest data

Direct Lake

- Loads data into VertiPaq engine in-memory

Direct Lake

– Loads data into VertiPaq engine in-memory

Direct Lake

- Loads data into VertiPaq engine in-memory

Direct Lake

–Loads data into VertiPaq engine in-memory

Direct Lake

- Loads data into VertiPaq engine in-memory
- **Transcoding:** Name for process by which one column is converted quickly from Parquet compressed columnar storage format into VertiPaq
    - Skips VertiPaq optimization: V-Order applies the sort optimization to Parquet
    - Inherits row-groups as segments
- **Reframing:** the word for refreshing a Direct Lake model: pointing it to a new snapshot
    - Evicts updated columns from RAM
    - Clears VertiPaq caches
- Only works on exact schema from Direct Lake: anything you might do in Power Query for an Import model must be done elsewhere

Direct Lake

- Skips VertiPaq optimization: V-Order applies the sort optimization to Parquet
  - Must write Parquet with Fabric-native components or post-process external Parquet
  - Still pay the compute for this V-Ordering; just amortize or shift when that compute happens
- Inherits row-groups as segments
  - Frequent-write and update-heavy patterns can yield an explosion of small segments
  - Requires VACUUM and table maintenance

Direct Lake

- Generally: a way to shift compute and refresh resources around
- Not magic
- Testing benefits is complex: requires deep understanding of data write and access patterns
- Think of it as an arbitrage opportunity for compute costs, not as a way to eliminate data maintenance
- Think of it as a way to **potentially** optimize refresh and **potentially not**

Refresh Scale-Out

Refresh Scale-Out

1. The Hidden Gem
2. How scale-out works
3. When to use it
4. What problem it solves
5. Why it’s not magic

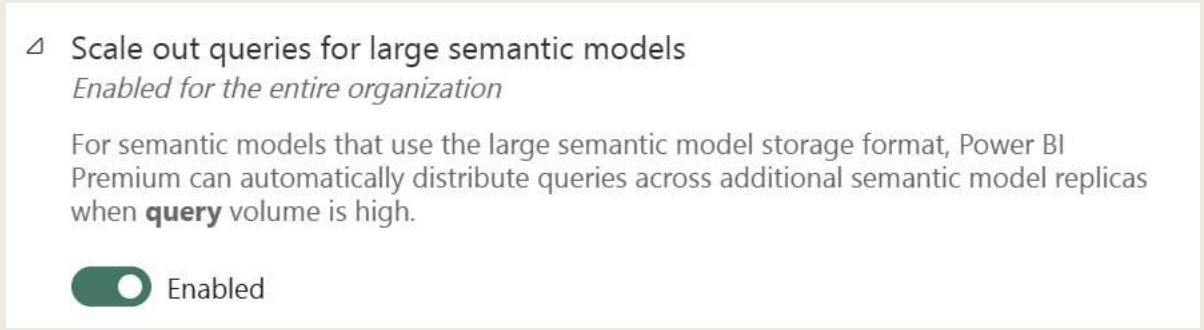
Refresh Scale-Out

- Officially known as Query scale-out, and has been available in Azure Analysis Services for as long as I can remember
- Consists of a processing node and multiple read-only query nodes, and is a feature for Semantic Models that are consumed by a large audience
- Can also be used to optimize memory while refreshing, as refresh and queries are separated on different nodes
- This allows doing a full refresh nearly to the memory limit, as you can Process Clear, followed by a Full Process, and then a Sync

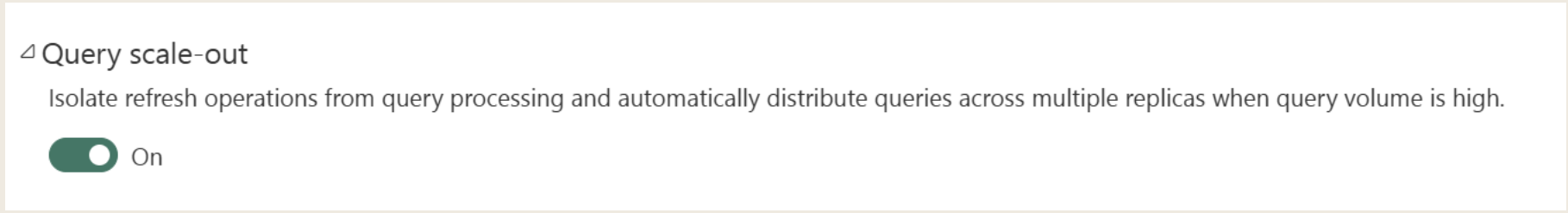
- *The individual semantic model replicas do not count against the max memory limitation per semantic model, and customers are not charged for the additional memory consumption of the semantic model replicas.*

Configure Refresh Scale-Out

- By default, it is enabled on the tenant level

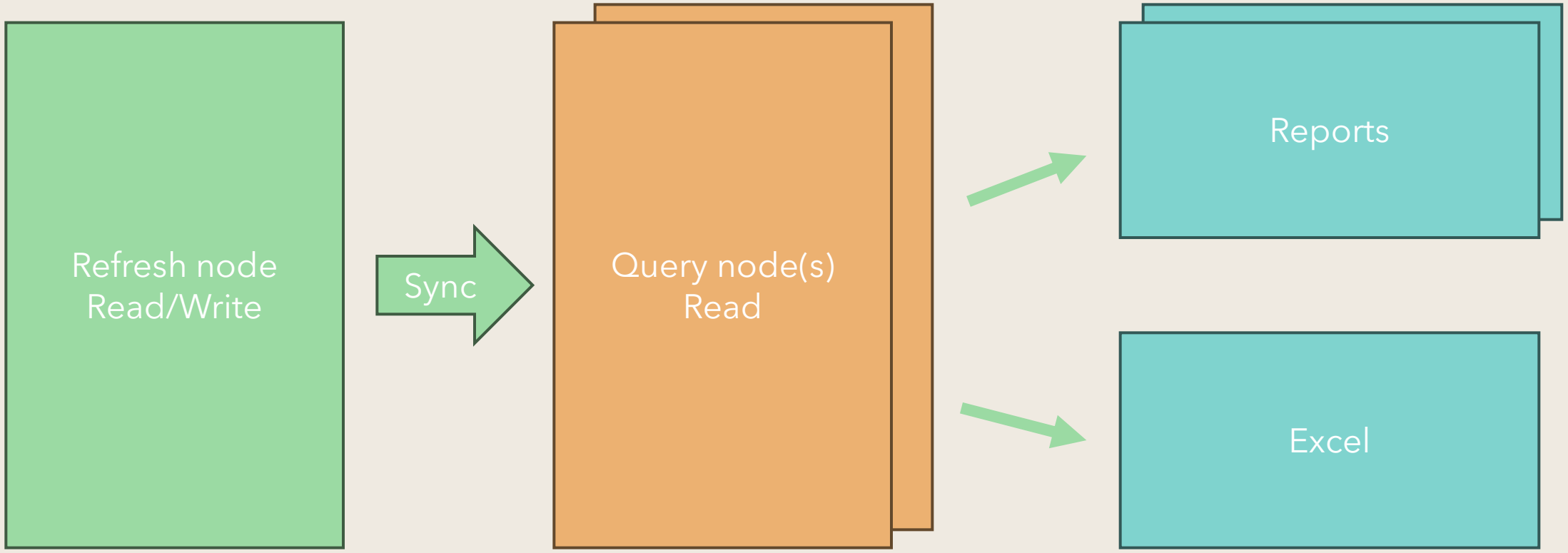


- Change the “maxReadOnlyReplicas” on the semantic model level from the default value of 0



Tabular Editor

Refresh Scale-Out



Wrap-up

Problem	Technique	Trade-off
Model too big	Reduce columns/cardinality	Less flexibility
Hierarchy overhead	Disable MDX	Excel limitation
Detail too large	Aggregations	DirectQuery latency
Refresh fails	Incremental / partitions / scale-out	Complexity

Resources

- Optimizing Semantic Model Memory in Fabric (tabulareditor.com/blog)

- A comprehensive guide to optimizing semantic model size (tabulareditor.com/blog)

- Building a Fabric Lab for Semantic Model Memory Experiments (justb.dk/blog)

Loved it? Learned something? Tell us! Share your feedback in just 1 minute

Tabular Editor

Tabular Editor

Thank you for your time!

A BIG thank you to the partners

Tabular Editor