Tabular Editor

Breaking the Limits: Advanced Optimization of Semantic Models in Microsoft Fabric



Data Saturday Holland 2026

Meet the speakers



Greg Baldini
Core Developer

Builds high-performance data solutions at Tabular Editor. Loves tackling big, hairy data problems – especially optimizing models. Talk to him about anything.

linkedin.com/in/gregbaldini/



Just Blindbæk
Field CTO

20 years in semantic modeling. Customer- and partner-facing at Tabular Editor; previously a consultant and trainer. SSAS Maestro and Microsoft MVP, Data Platform.

linkedin.com/in/blindbaek/

Tabular Editor

Meet the sponsors!



























Agenda

1. Make the model smaller

   Fabric memory limits · VertiPaq internals · Model size diagnosis · Cardinality · Aggregations

2. Make the model process reliable

   Peak memory · Partitions · Incremental refresh · Hybrid tables · Direct Lake · Scale-out

3. Make the model query efficiently

   Storage Engine vs Formula Engine · Relationships · DAX/query patterns · Server Timings · Query plans

Throughout the session: diagnose → optimize → measure → repeat

Understand where the memory goes before trying to reduce it

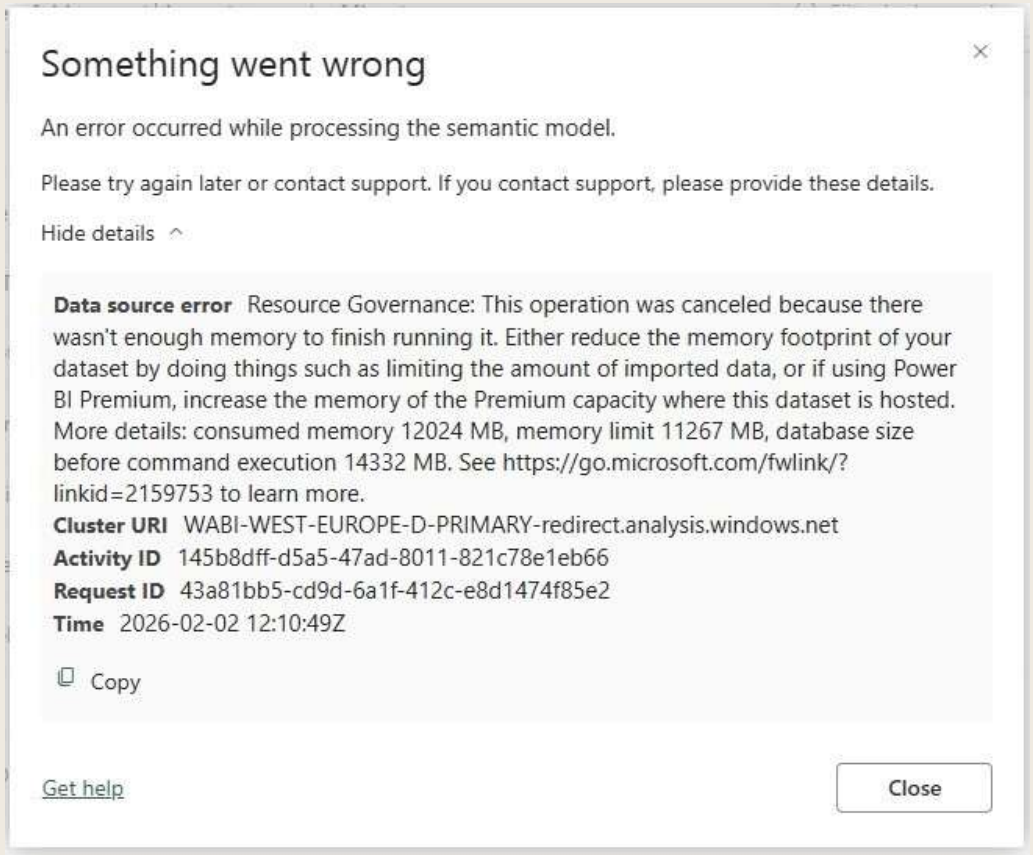
Make the model smaller

Fabric Capacity Limits

Fabric Capacity Limits

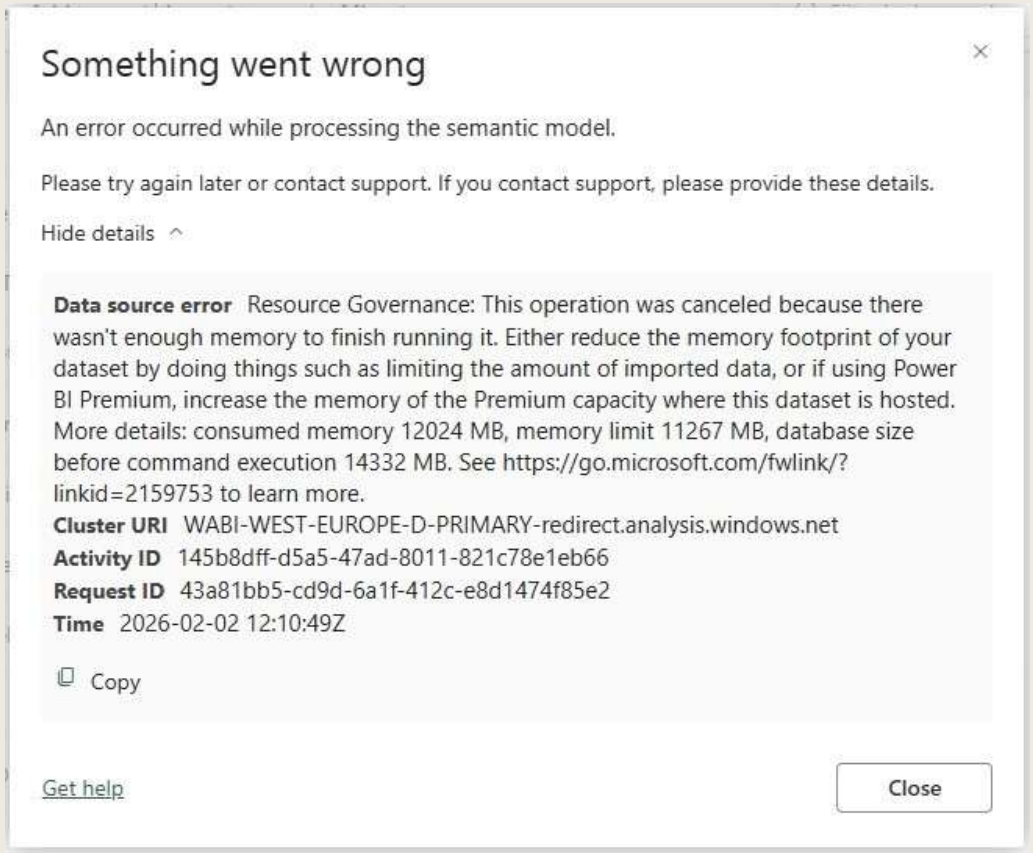
1. Fabric Capacities Are Powerful – Memory Is a Bottleneck
2. Fabric SKU Memory Limits (Model Size)
3. Model Size ≠ Refresh Memory Usage
4. The Design Choice - You Always Pay Somewhere
5. Where Does the Memory Go?

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.

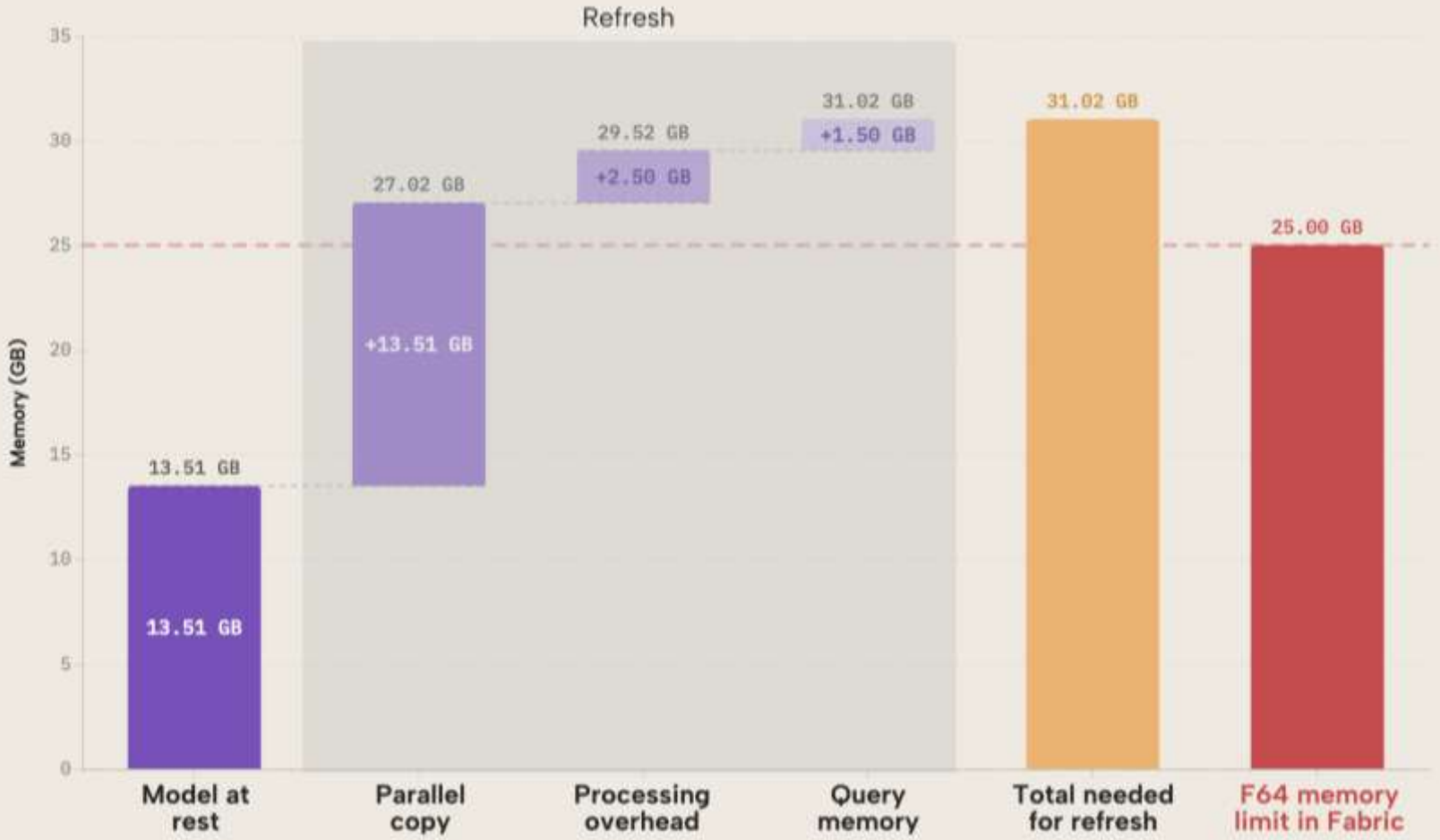
Fabric SKU Memory Limits (Model Size)

- 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
- No auto-scale available
- 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
F256	100 GB

SKU memory limits are enforced per semantic model

Steady-State vs Peak Memory



Tabular Editor

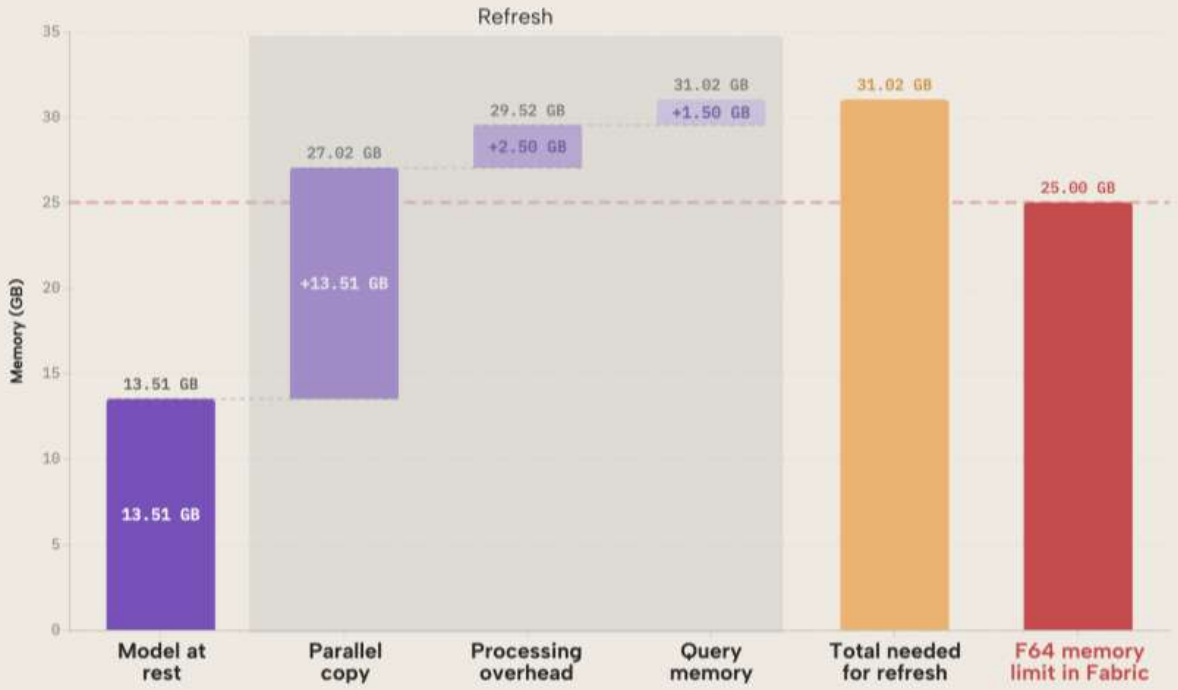
Steady-State vs Peak Memory

Steady-State Memory

- Model in memory
- User queries

Peak Memory (Refresh)

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



Tabular Editor

What problem are we actually solving?

Model too large

- Steady-state memory

Refresh fails

- Peak processing memory

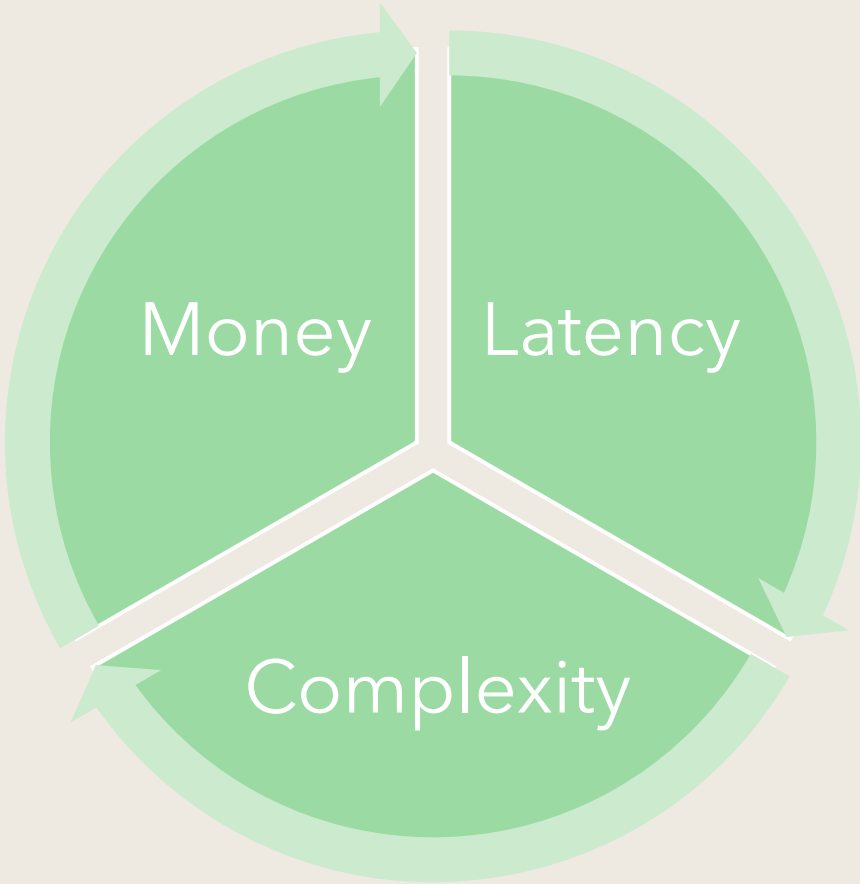
Queries are slow

- Model / DAX / engine behavior

Different symptoms. Different fixes. Diagnose first.

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

Meet our lab: NYC Yellow Taxi

- Scales from hundreds of millions to billions of rows depending on the years loaded
- 194.5M rows in the fact table
- 13.5 GiB total model size
- 66 columns across 8 tables
- Built from the public NYC Yellow Taxi trip data
- Designed to expose real semantic-model memory and performance challenges
- Used throughout today's demos

Build the lab yourself

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

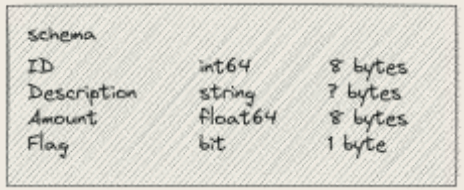
1. Columnstore
2. Compression
3. Hierarchies
4. Compression (RLE hint)
5. Processing memory vs steady-state memory

Tighten this section.

Data storage: View for humans

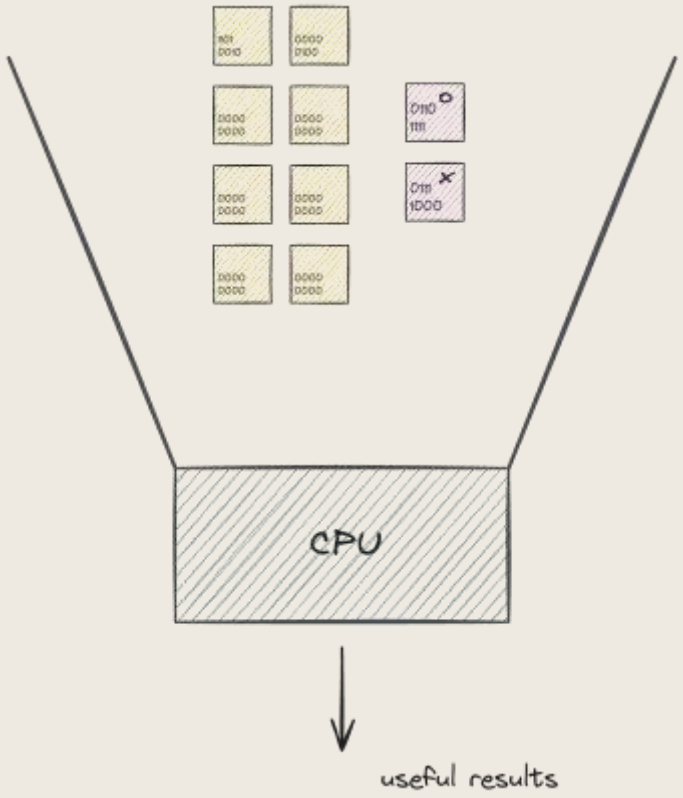
- 4 columns
- 3 types
- 10 rows

ID	Description	Amount	Flag
2	ox	-2832.21	0
8	dinosaur	33289.8	1
7	pen	43987	1
5	hippopotamus	92348.1	1
1	cat	7	0
3	sesquipedalian	232.832	1
4	microscope	0.00984	1
6	rhinoceros	2048	0
9	daffodil	-17.5	1
10	electroencephalograph	489000	1



Tabular Editor

Hardware-software systems in one slide

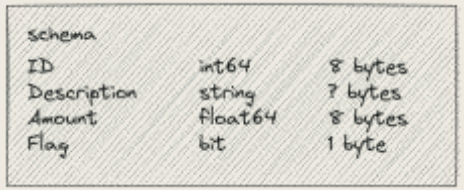


- Performance: shovel data through the CPU as fast as you can
  - CPU always sees a *cache line:* 64 bytes
- Operating system provides access to hardware
  - Disk and RAM: managed as *pages* (>=4KiB chunks)
  - Also: network, GPU, peripherals
- Modern systems can process 100s of GiB/s
  - Unless you mess things up for them (;

Tabular Editor

Data storage: View for CPUs

ID	Description	Amount	Flag
2	ox	-2832.21	0
8	dinosaur	33289.8	1
7	pen	43987	1
5	hippopotamus	92348.1	1
1	cat	7	0
3	sesquipedalian	232.832	1
4	microscope	0.00984	1
6	rhinoceros	2048	0
9	daffodil	-17.5	1
10	electroencephalograph	489000	1



- 4 columns
- 3 types
- 10 rows

- How do we store it?

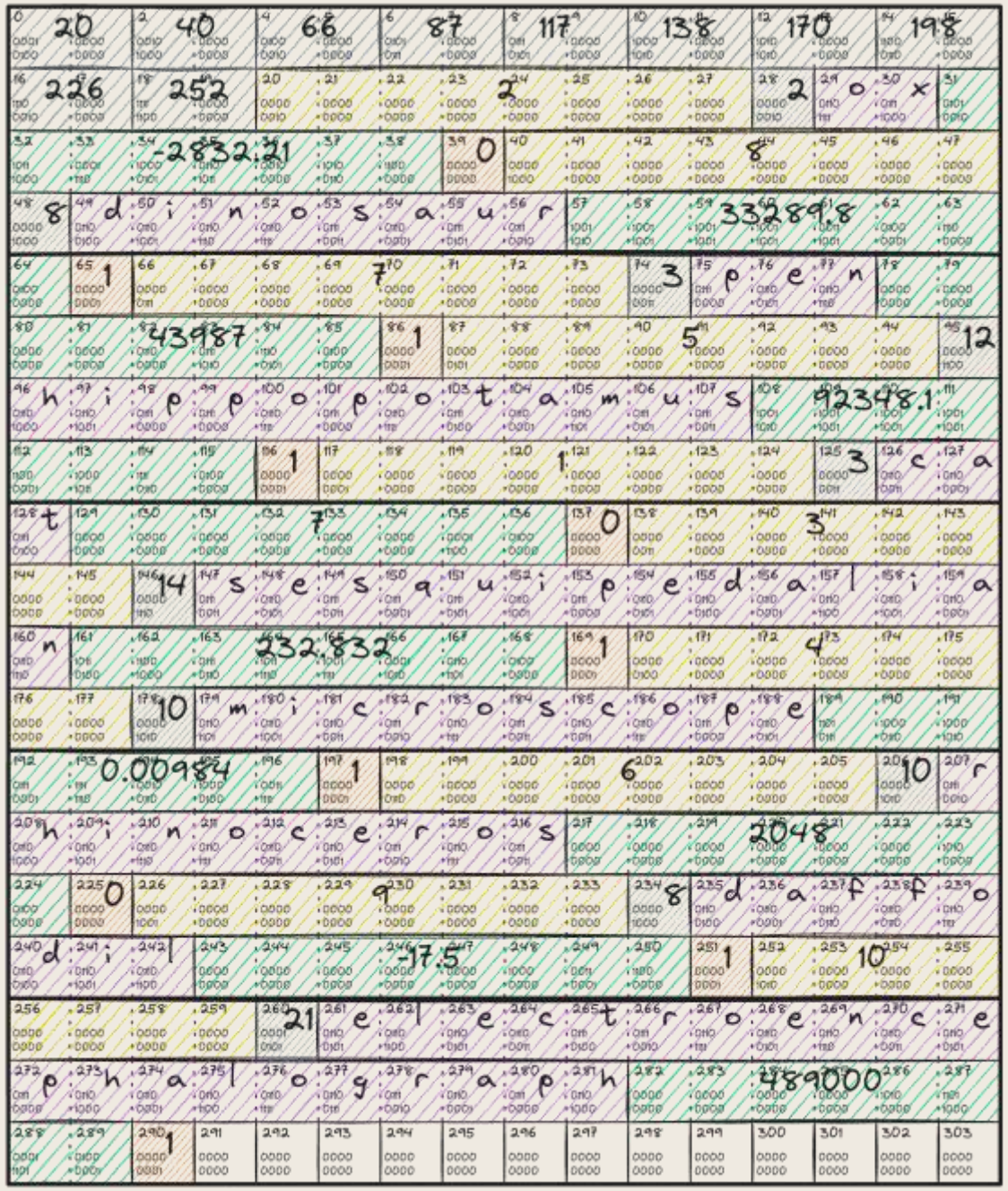
Tabular Editor

Data storage: row-based

ID	Description	Amount	Flag
2	ox	-2832.21	0
8	dinosaur	33289.8	1
7	pen	43987	1
5	hippopotamus	92348.1	1
1	cat	7	0
3	sesquipedalian	232.832	1
4	microscope	0.00984	1
6	rhinoceros	2048	0
9	daffodil	-17.5	1
10	electroencephalograph	489000	1

schema

ID	int64	8 bytes
Description	string	? bytes
Amount	float64	8 bytes
Flag	bit	1 byte



1 cell = 1 byte; heavy rule = cache line boundary; page continues to 4 KiB

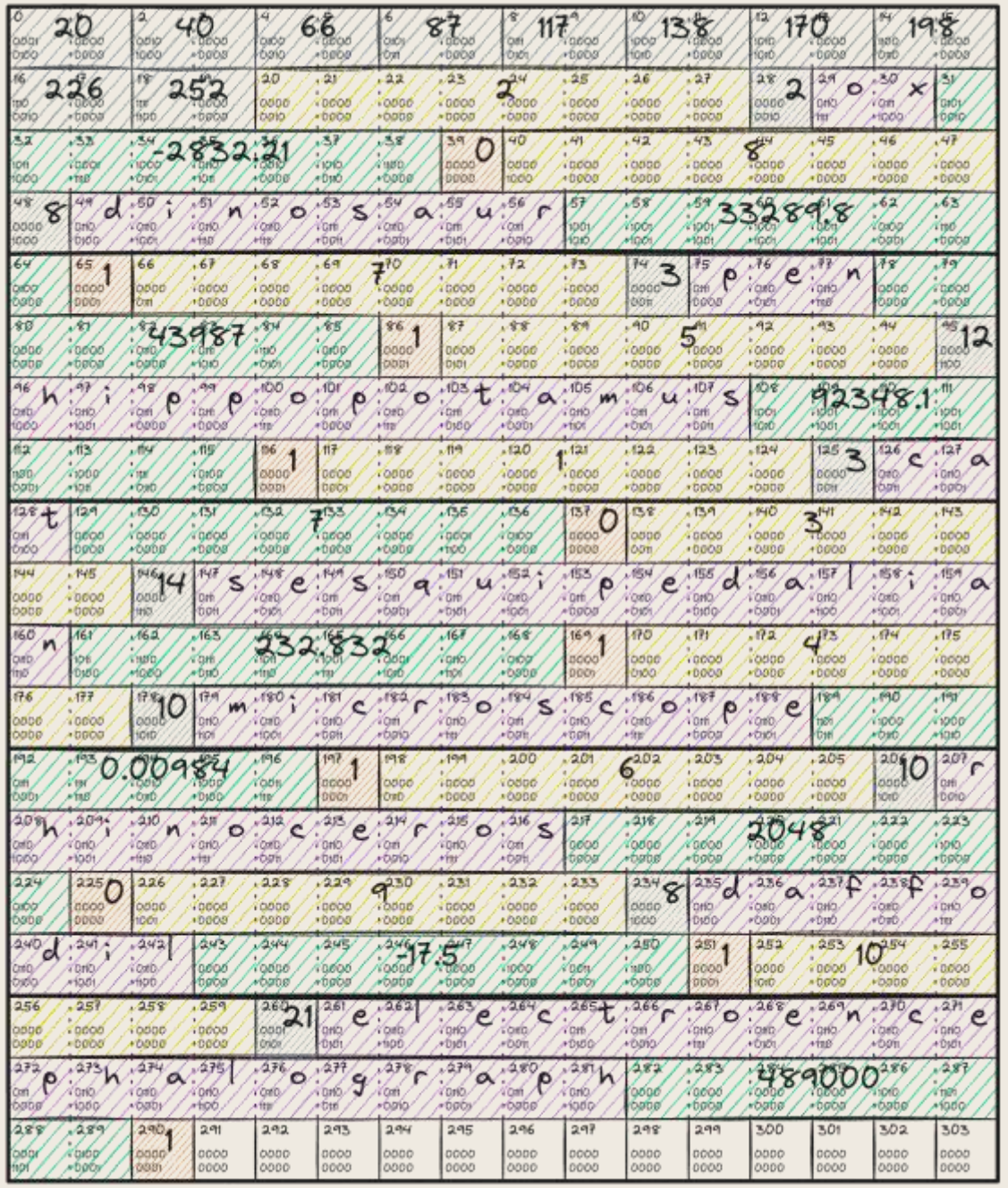
Tabular Editor

Data storage: row-based

ID	Description	Amount	Flag
2	ox	-2832.21	0
8	dinosaur	33289.8	1
7	pen	43987	1
5	hippopotamus	92348.1	1
1	cat	7	0
3	sesquipedalian	232.832	1
4	microscope	0.00984	1
6	rhinoceros	2048	0
9	daffodil	-17.5	1
10	electroencephalograph	489000	1

schema

ID	int64	8 bytes
Description	string	? bytes
Amount	float64	8 bytes
Flag	bit	1 byte



1 cell = 1 byte; heavy rule = cache line boundary; page continues to 4 KiB

Header, offsets pointing to starts of rows

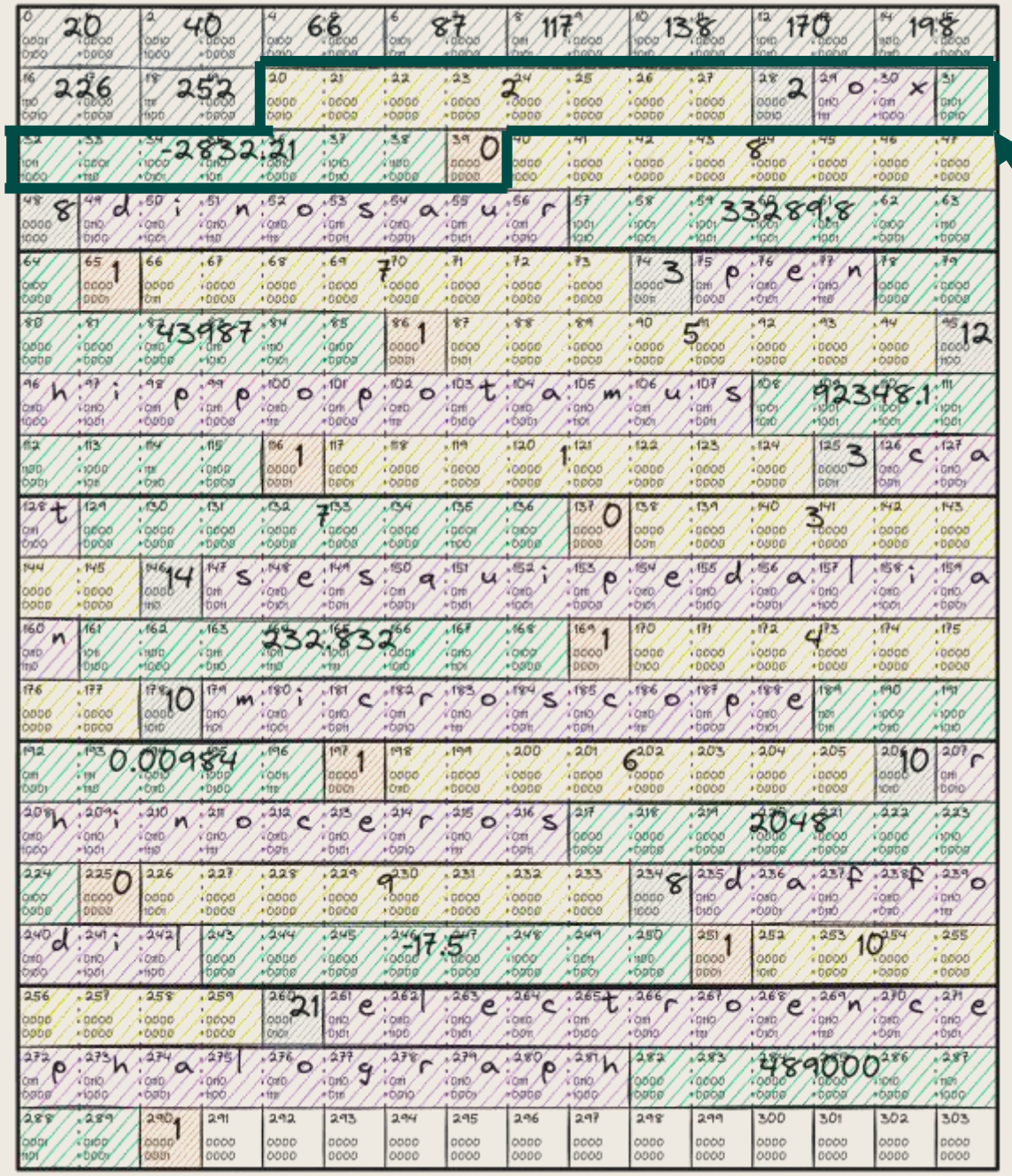
Tabular Editor

Data storage: row-based

ID	Description	Amount	Flag
2	ox	-2832.21	0
8	dinosaur	33289.8	1
7	pen	43987	1
5	hippopotamus	92348.1	1
1	cat	7	0
3	sesquipedalian	232.832	1
4	microscope	0.00984	1
6	rhinoceros	2048	0
9	daffodil	-17.5	1
10	electroencephalograph	489000	1

schema

ID	int64	8 bytes
Description	string	? bytes
Amount	float64	8 bytes
Flag	bit	1 byte



1 cell = 1 byte; heavy rule = cache line boundary; page continues to 4 KiB

Header, offsets pointing to starts of rows

The first row, at offset 20

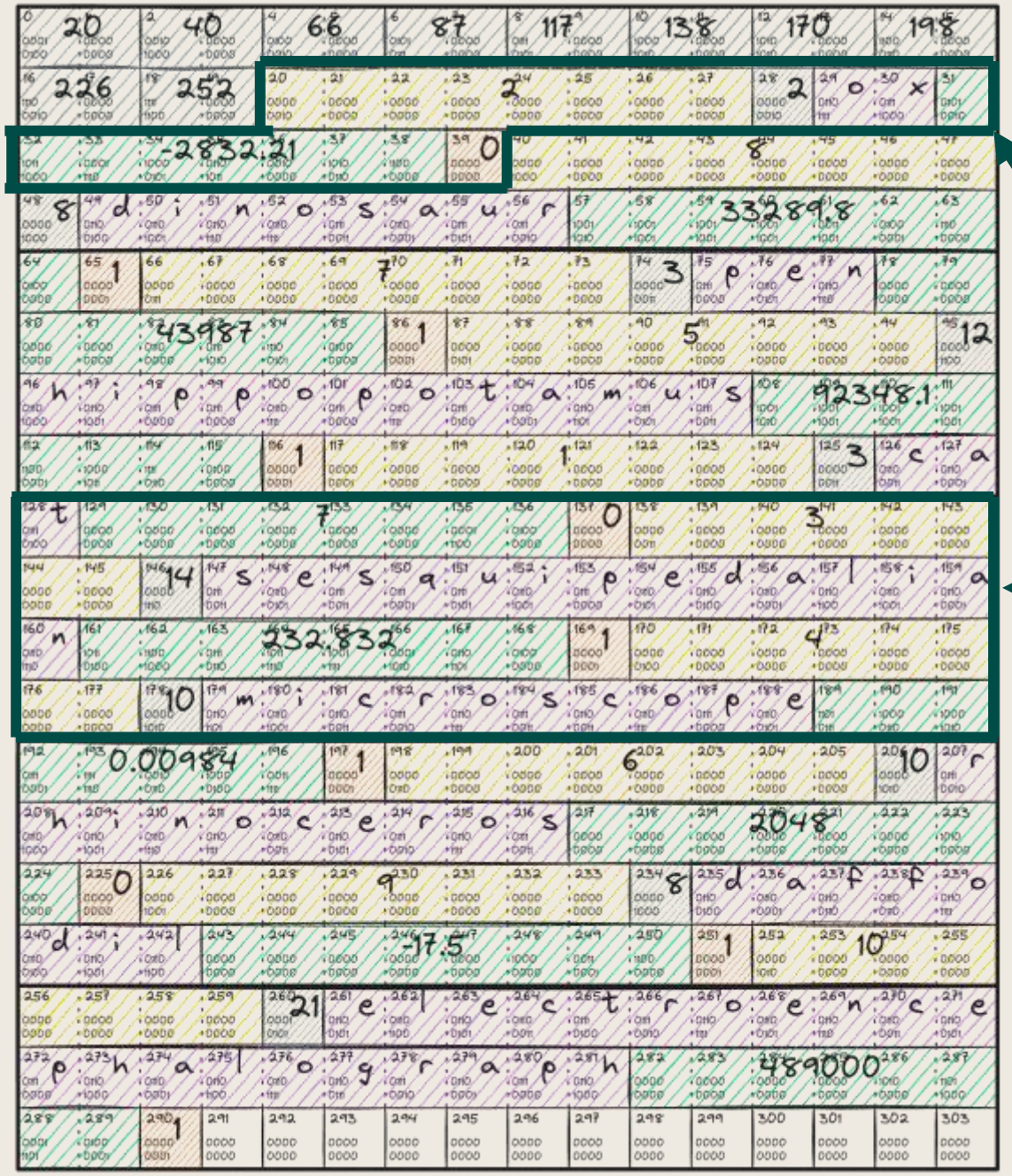
Tabular Editor

Data storage: row-based

ID	Description	Amount	Flag
2	ox	-2832.21	0
8	dinosaur	33289.8	1
7	pen	43987	1
5	hippopotamus	92348.1	1
1	cat	7	0
3	sesquipedalian	232.832	1
4	microscope	0.00984	1
6	rhinoceros	2048	0
9	daffodil	-17.5	1
10	electroencephalograph	489000	1

schema

ID	int64	8 bytes
Description	string	? bytes
Amount	float64	8 bytes
Flag	bit	1 byte



1 cell = 1 byte; heavy rule = cache line boundary; page continues to 4 KiB

Header, offsets pointing to starts of rows

The first row, at offset 20

A cache line: what the CPU reads at once

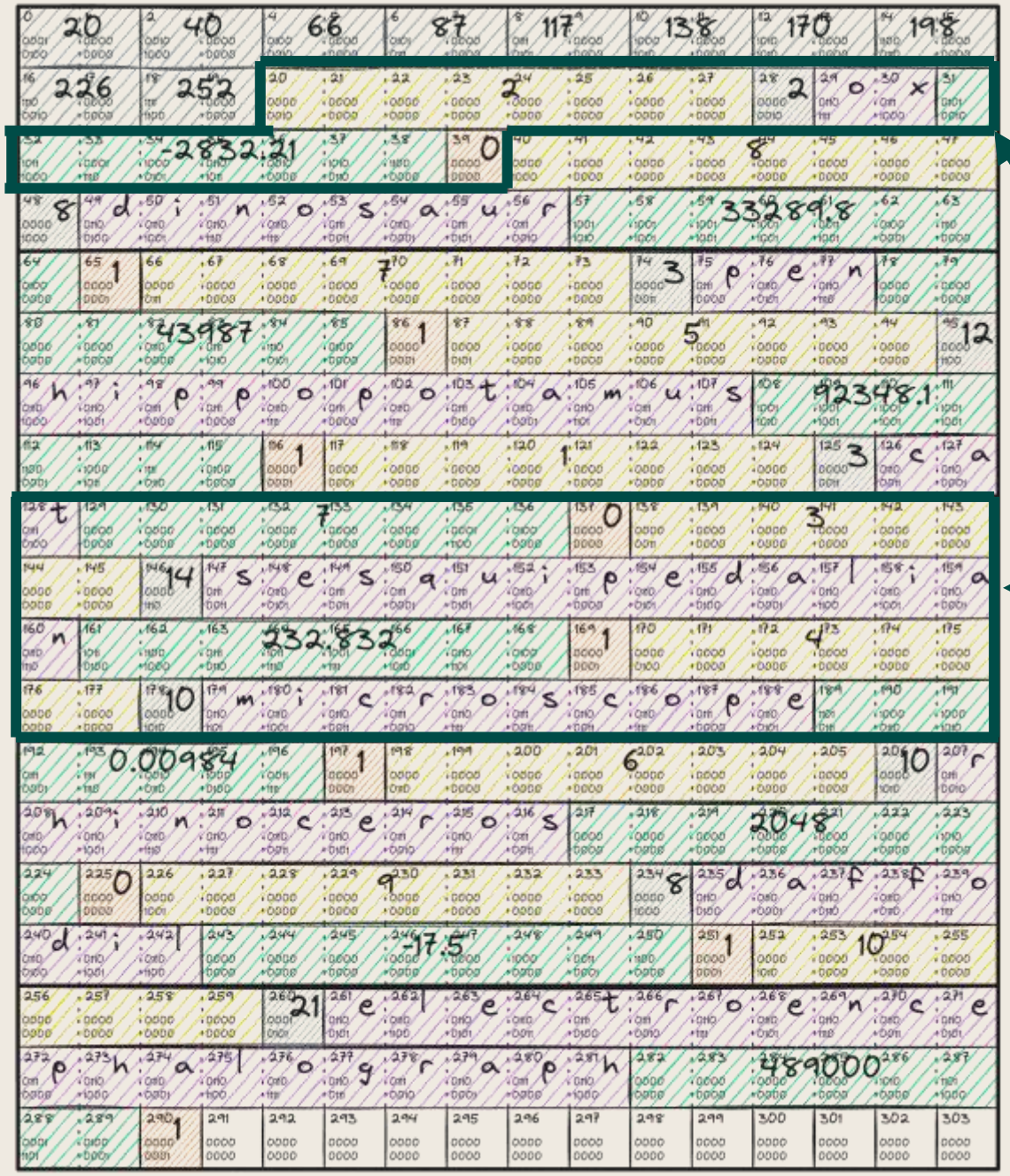
Tabular Editor

Data storage: row-based

ID	Description	Amount	Flag
2	ox	-2832.21	0
8	dinosaur	33289.8	1
7	pen	43987	1
5	hippopotamus	92348.1	1
1	cat	7	0
3	sesquipedalian	232.832	1
4	microscope	0.00984	1
6	rhinoceros	2048	0
9	daffodil	-17.5	1
10	electroencephalograph	489000	1

schema

ID	int64	8 bytes
Description	string	? bytes
Amount	float64	8 bytes
Flag	bit	1 byte



1 cell = 1 byte; heavy rule = cache line boundary; page continues to 4 KiB

Header, offsets pointing to starts of rows

The first row, at offset 20

A cache line: what the CPU reads at once

Elsewhere: more metadata, incl. the schema

Tabular Editor

Rowstore

- Header page(s) with aggregate metadata about a whole table
- Header in every page describing data in page (row positions, + more)
- Pages hold collection of rows
- Variable field lengths make all data positions unpredictable

Data storage: column-based

ID	Description	Amount	Flag
2	ox	-2832.21	0
8	dinosaur	33289.8	1
7	pen	43987	1
5	hippopotamus	92348.1	1
1	cat	7	0
3	sesquipedalian	232.832	1
4	microscope	0.00984	1
6	rhinoceros	2048	0
9	daffodil	-17.5	1
10	electroencephalograph	489000	1

schema

ID	int64	8 bytes
Description	string	? bytes
Amount	float64	8 bytes
Flag	bit	1 byte

ID page



1 cell = 1 byte; heavy rule = cache line boundary; page continues to 4 KiB

Tabular Editor

Data storage: column-based

ID	Description	Amount	Flag
2	ox	-2832.21	0
8	dinosaur	33289.8	1
7	pen	43987	1
5	hippopotamus	92348.1	1
1	cat	7	0
3	sesquipedalian	232.832	1
4	microscope	0.00984	1
6	rhinoceros	2048	0
9	daffodil	-17.5	1
10	electroencephalograph	489000	1

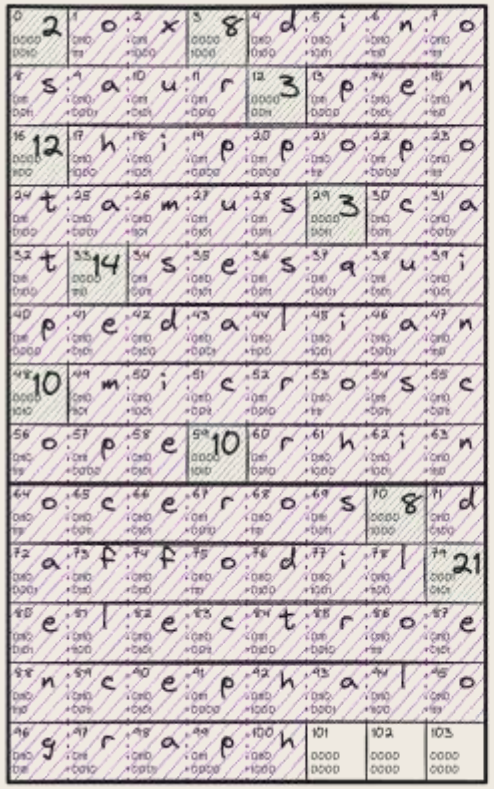
schema		
ID	int64	8 bytes
Description	string	? bytes
Amount	float64	8 bytes
Flag	bit	1 byte

ID page



1 cell = 1 byte; heavy rule = cache line boundary; page continues to 4 KiB

Description page



1 cell = 1 byte; heavy rule = cache line boundary; page continues to 4 KiB

Tabular Editor

Data storage: column-based

ID	Description	Amount	Flag
2	ox	-2832.21	0
8	dinosaur	33289.8	1
7	pen	43987	1
5	hippopotamus	92348.1	1
1	cat	7	0
3	sesquipedalian	232.832	1
4	microscope	0.00984	1
6	rhinoceros	2048	0
9	daffodil	-17.5	1
10	electroencephalograph	489000	1

schema

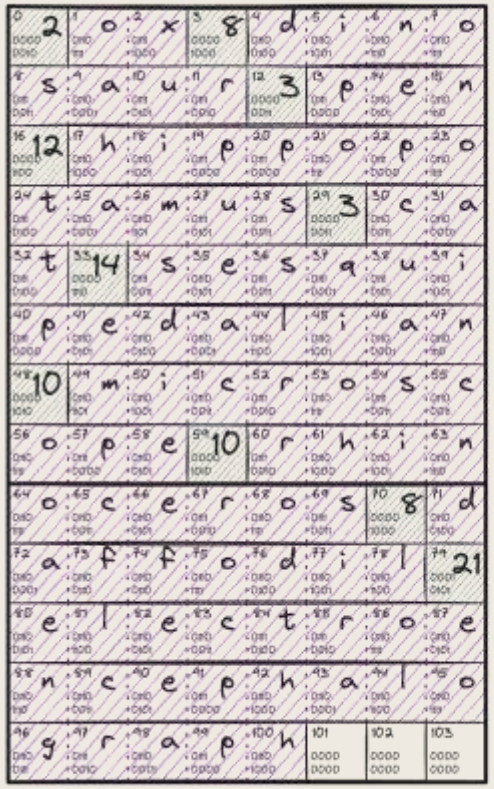
ID	int64	8 bytes
Description	string	? bytes
Amount	float64	8 bytes
Flag	bit	1 byte

ID page



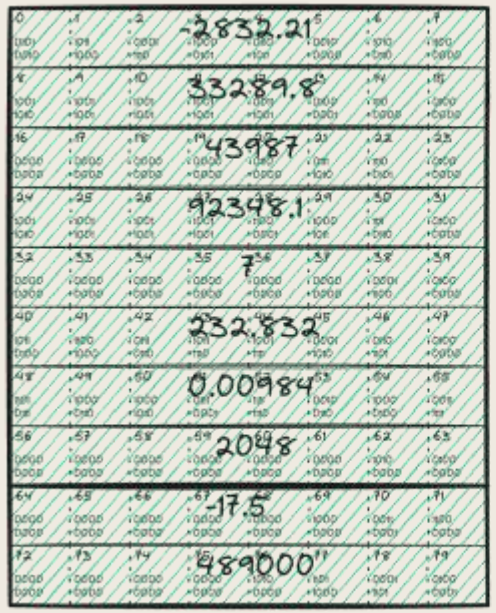
1 cell = 1 byte; heavy rule = cache line boundary; page continues to 4 KiB

Description page



1 cell = 1 byte; heavy rule = cache line boundary; page continues to 4 KiB

Amount page



1 cell = 1 byte; heavy rule = cache line boundary; page continues to 4 KiB

Tabular Editor

Data storage: column-based

ID	Description	Amount	Flag
2	ox	-2832.21	0
8	dinosaur	33289.8	1
7	pen	43987	1
5	hippopotamus	92348.1	1
1	cat	7	0
3	sesquipedalian	232.832	1
4	microscope	0.00984	1
6	rhinoceros	2048	0
9	daffodil	-17.5	1
10	electroencephalograph	489000	1

schema

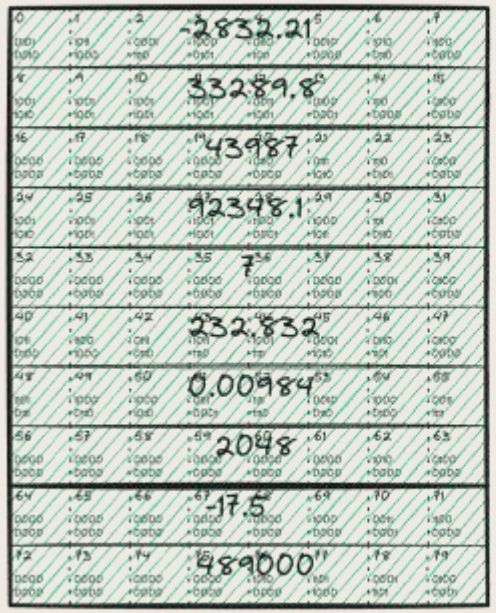
ID	int64	8 bytes
Description	string	? bytes
Amount	float64	8 bytes
Flag	bit	1 byte

ID page



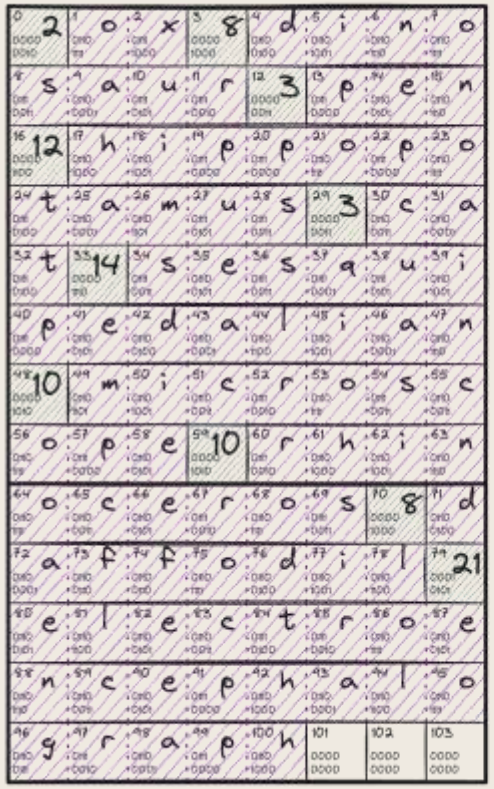
1 cell = 1 byte; heavy rule = cache line boundary; page continues to 4 KiB

Amount page



1 cell = 1 byte; heavy rule = cache line boundary; page continues to 4 KiB

Description page



1 cell = 1 byte; heavy rule = cache line boundary; page continues to 4 KiB

Flag page



1 cell = 1 byte; heavy rule = cache line boundary; page continues to 4 KiB

Tabular Editor

Data storage: column-based

ID	Description	Amount	Flag
2	ox	-2832.21	0
8	dinosaur	33289.8	1
7	pen	43987	1
5	hippopotamus	92348.1	1
1	cat	7	0
3	sesquipedalian	232.832	1
4	microscope	0.00984	1
6	rhinoceros	2048	0
9	daffodil	-17.5	1
10	electroencephalograph	489000	1

schema

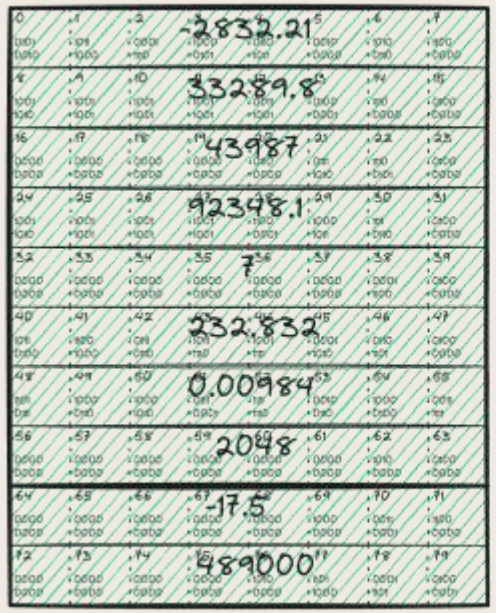
ID	int64	8 bytes
Description	string	? bytes
Amount	float64	8 bytes
Flag	bit	1 byte

ID page



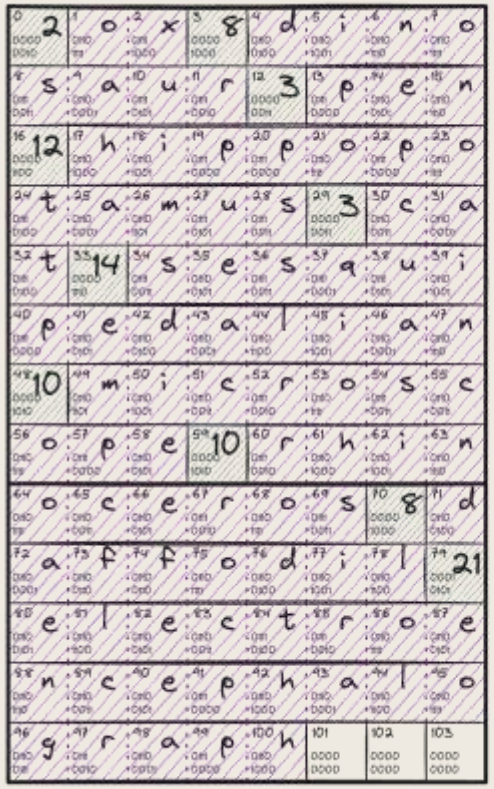
1 cell = 1 byte; heavy rule = cache line boundary; page continues to 4 KiB

Amount page



1 cell = 1 byte; heavy rule = cache line boundary; page continues to 4 KiB

Description page



1 cell = 1 byte; heavy rule = cache line boundary; page continues to 4 KiB

Flag page



1 cell = 1 byte; heavy rule = cache line boundary; page continues to 4 KiB

Elsewhere:
more
metadata, incl.
the schema

Tabular Editor

Columnstore

- Header page(s) with aggregate metadata (~same as rowstore)
- Header in every page describing data in page (same idea, but different metadata to rowstore)
- **Pages hold collection of values from a single column**
- **Fixed-length fields (almost all numeric types, bits / logical, dates) have known positions in the page**

VertiPaq is a compressed column-store engine

Compression

- Value encoding
- Dictionary / hash
- Run-length

All operate on *segments*: 1M+ records at a time

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

Tabular Editor

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

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

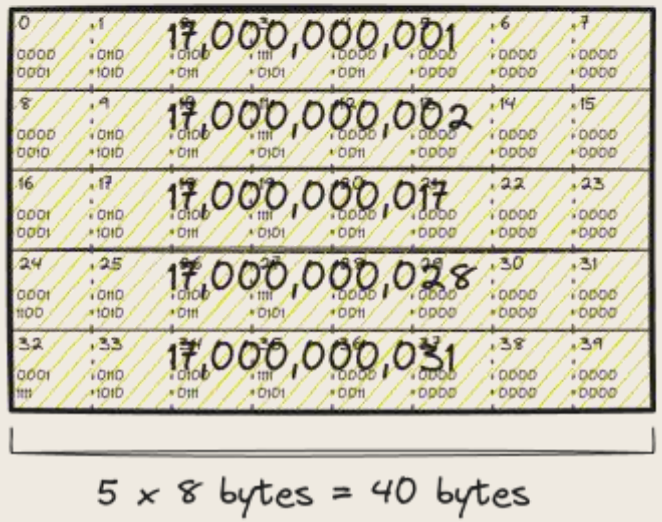


Tabular Editor

Value encoding

raw

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



Some questions:

What is the same?
What is different?

How big is the common part?
How big is the varying part?

Tabular Editor

Value encoding

raw

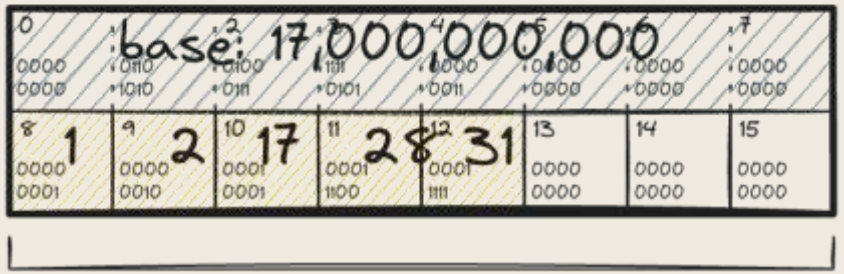
Number
17,000,000,001
17,000,000,002
17,000,000,017
17,000,000,028
17,000,000,031



5 x 8 bytes = 40 bytes

value encoded

Offset
1
2
17
28
31



8 + 5 x 1 = 13 bytes

Tabular Editor

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

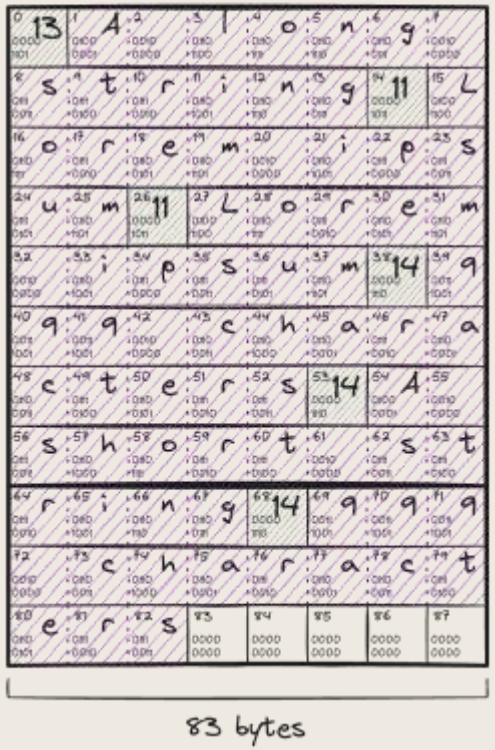


Tabular Editor

Dictionary encoding

raw

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



Some questions:

What is the same?
What is different?

How big is the common part?

Tabular Editor

Dictionary encoding

raw

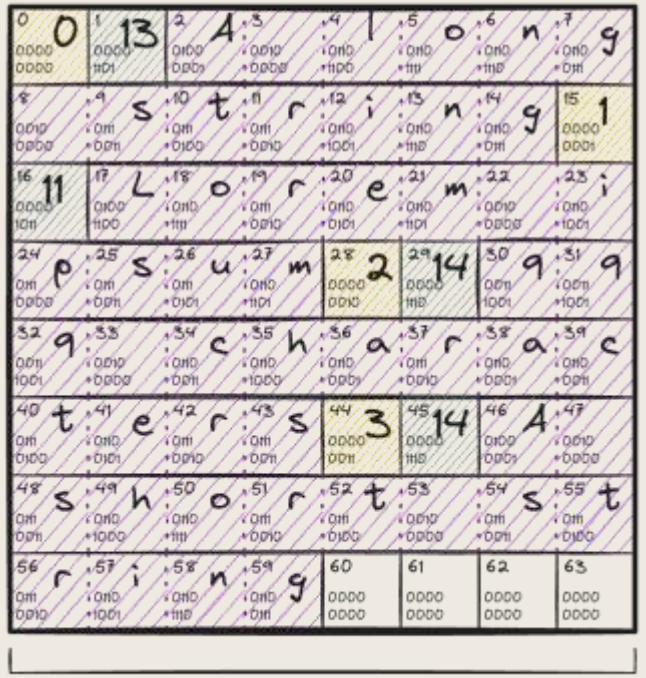
String
A long string
Lorem ipsum
Lorem ipsum
999 characters
A short string
999 characters



83 bytes

dictionary

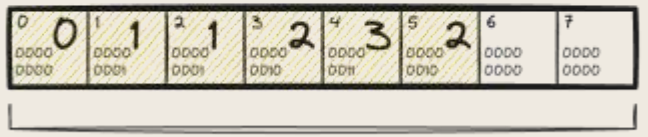
DictKey	String
0	A long string
1	Lorem ipsum
2	999 characters
3	A short string



60 bytes

encoded column

DictKey
0
1
1
2
3
2



6 bytes

Tabular Editor

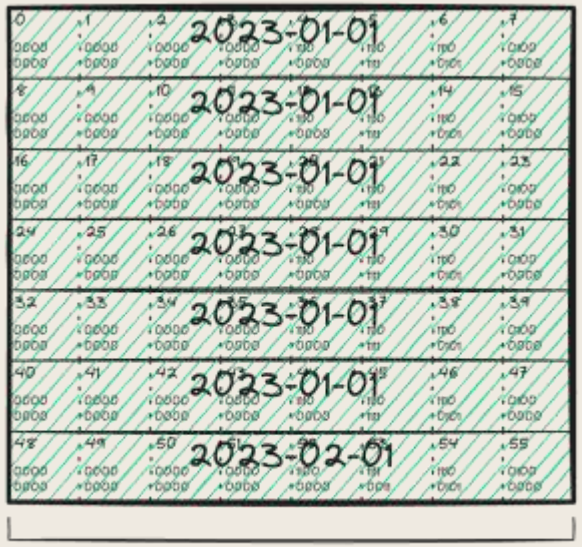
Dictionary encoding

- 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
2023-02-01

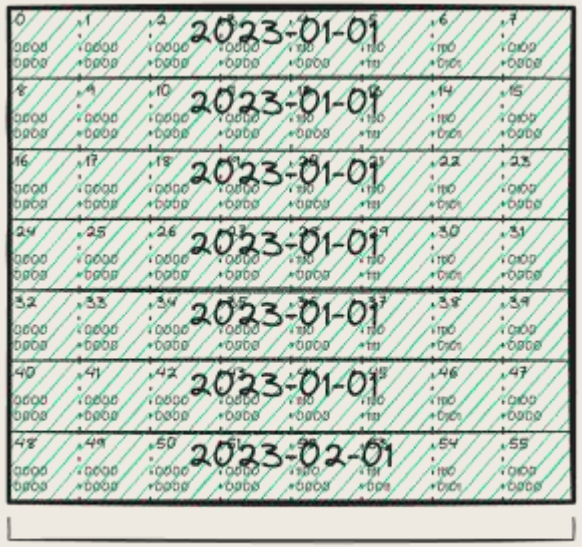


Tabular Editor

Run-length encoding

raw

Date
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-02-01



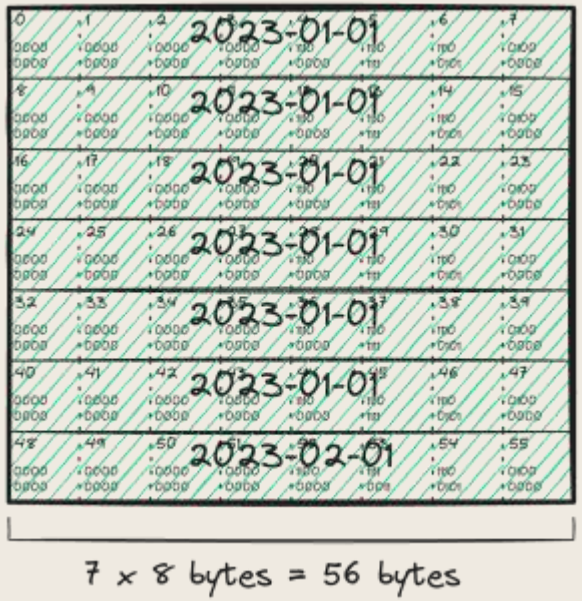
You know the drill….

Tabular Editor

Run-length encoding

raw

Date
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-02-01



You know the drill….

What is the same?
What is different?

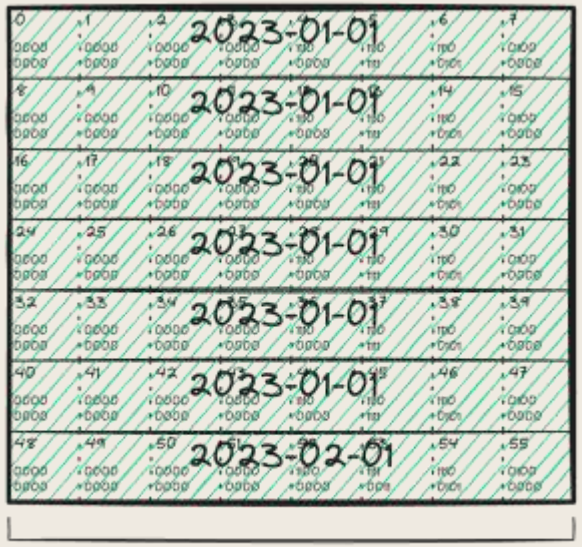
How big is the common part?

Tabular Editor

Run-length encoding

raw

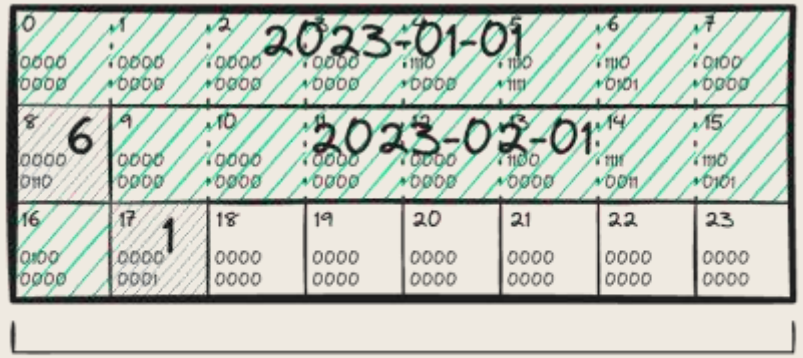
Date
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-02-01



7 x 8 bytes = 56 bytes

run-length encoded

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



2 x (8 + 1) = 18 bytes

Tabular Editor

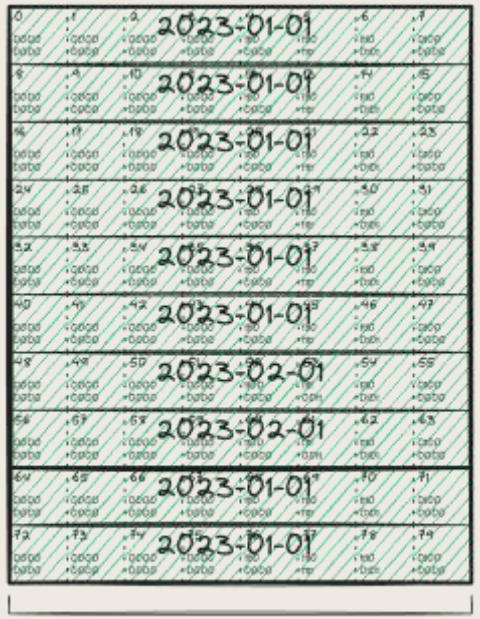
Run-length encoding

- 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

One big (small) happy compression family

raw

Date
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-02-01
2023-02-01
2023-01-01
2023-01-01

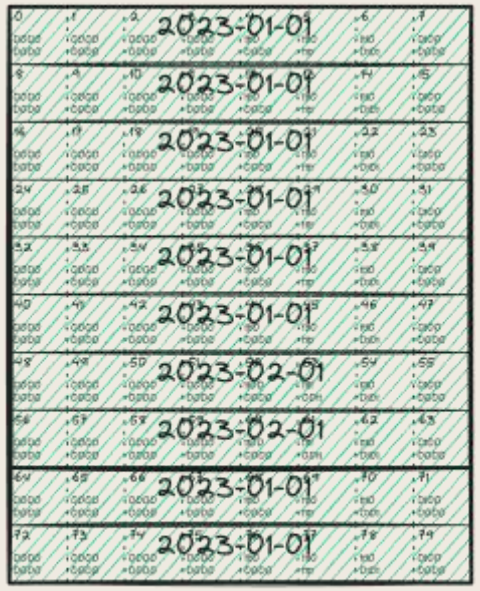


Tabular Editor

One big (small) happy compression family

raw

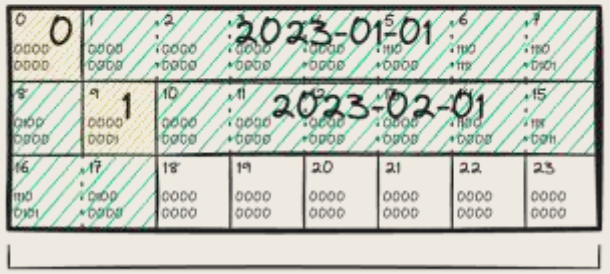
Date
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-02-01
2023-02-01
2023-01-01
2023-01-01



10 x 8 bytes = 80 bytes

dictionary

DictKey	Date
0	2023-01-01
1	2023-02-01



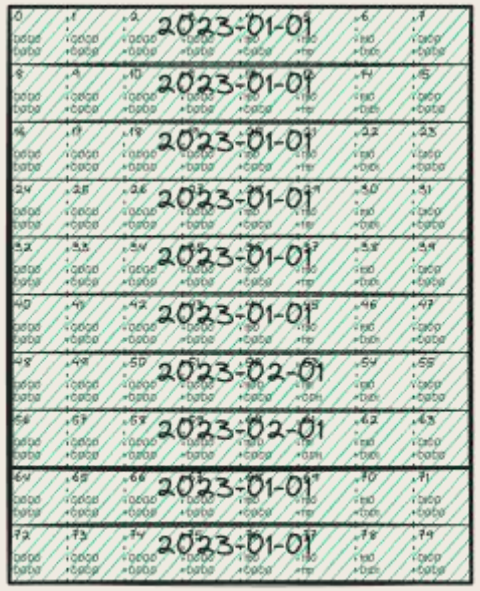
18 bytes

Tabular Editor

One big (small) happy compression family

raw

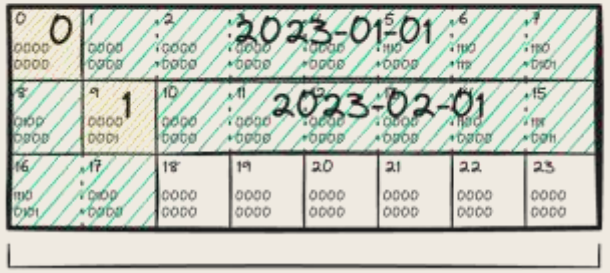
Date
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-01-01
2023-02-01
2023-02-01
2023-01-01
2023-01-01



10 x 8 bytes = 80 bytes

dictionary

DictKey	Date
0	2023-01-01
1	2023-02-01



18 bytes

runs of keys

DictKey	Count
0	6
1	2
0	2



6 bytes

Dictionary encoding to minimal bit keys + run-length encoding of the keys

Tabular Editor

Recap

- Cardinality is primary determinant of compression ratio
- Larger segments offer better opportunities for high compression ratios
- Primary column size components
    - Dictionary
    - Data
    - Hierarchies

Diagnosing model size

Diagnosing model size

1. Diagnose before you optimize
2. Tools for model-size diagnosis

Diagnose before you optimize

- Start with evidence, not assumptions
- Identify the largest tables and columns
- Separate **Data**, **Dictionary**, and **Hierarchy** size
- Look for high-cardinality columns and technical attributes
- Ask whether each expensive column is actually needed
- Measure before and after every change

Tools for model-size diagnosis

Tabular Editor 3

- VertiPaq Analyzer
- Table & column size
- Cardinality & encoding
- Data / Dictionary / Hierarchy
- Inspect and change model metadata

DAX Studio

- VertiPaq Analyzer
- DMV / metadata inspection
- Detailed storage statistics
- Export and deeper analysis
- Later: Server Timings & query plans

Fabric / Power BI Service

- Model and SKU memory limits
- Refresh history and failures
- Capacity context
- Capacity Metrics
- Understand the environment around the model

Use the tool that answers the question you currently have.



Interactive challenge #1

- Yellow Taxi
  - 194.5M fact rows
  - 13.5 GiB semantic model
  - 66 columns across 8 tables
  - F64 capacity

Your model is too large. What do you inspect first?



Interactive challenge #1

- Upgrade the capacity, or change the model?

  A. Upgrade the capacity

  B. Look for the largest tables

  C. Look for the highest-cardinality columns

  D. Check dictionary and hierarchy size

  E. Start deleting columns you think are unnecessary

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

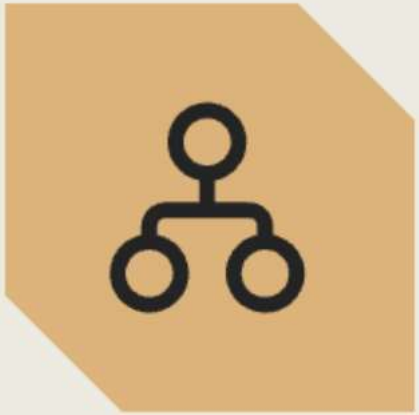
Tabular Editor

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.

Measure the result

- Recalculate model size after each change
- Compare before vs. after
- Identify which change delivered the biggest gain
- Check whether query performance also changed
- Validate that functionality was not lost
- Keep the trade-off visible

Optimize the biggest contributors first – then measure what you gained.

Strategic Detail Isolation

Strategic Detail Isolation

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

Remove historic details

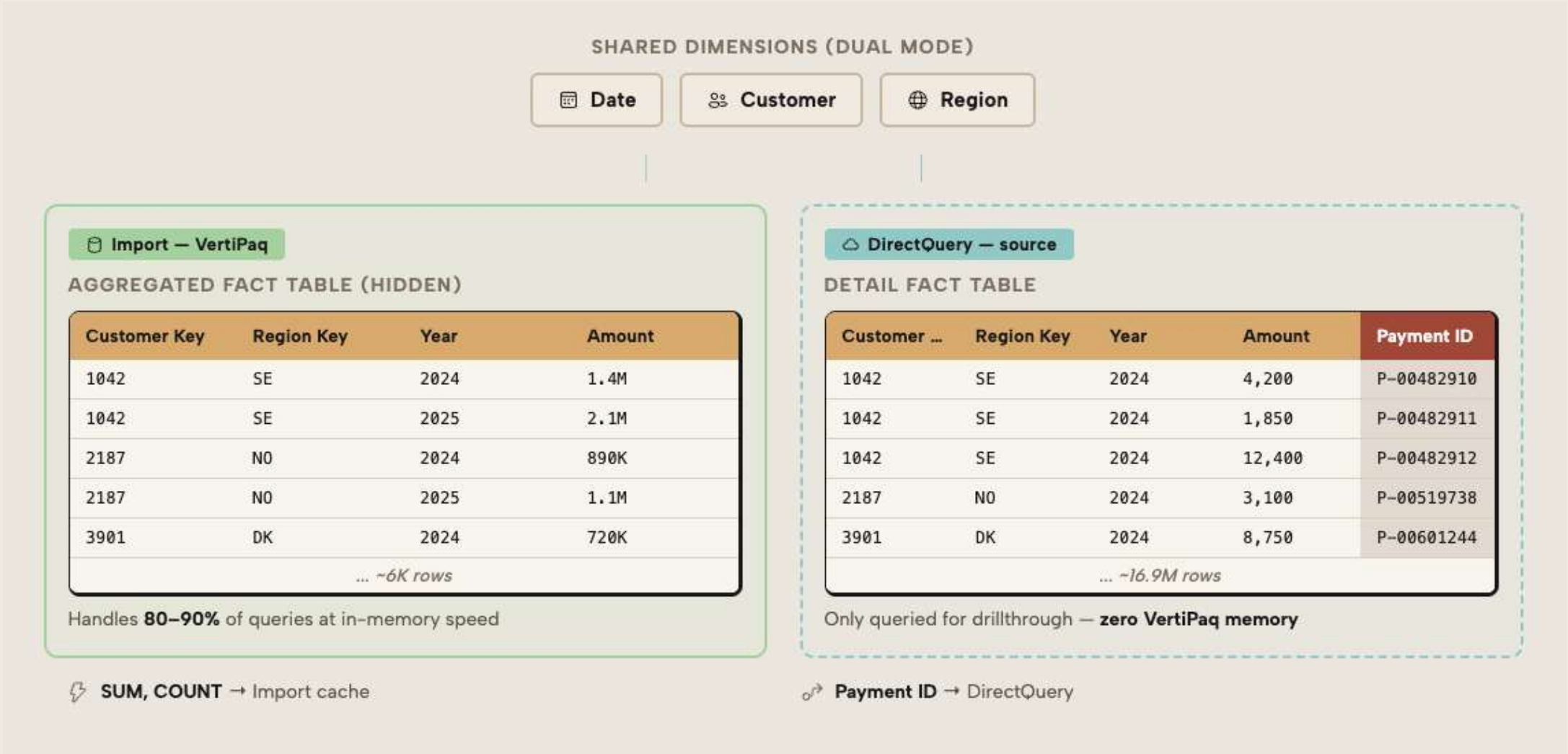
- Removing high-cardinality columns is ALWAYS the first choice
- Cannot remove the column?
  - Then, only include data in the column for the last year?
  - This most likely limits the cardinality

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



Tabular Editor

When not to import everything

- Detail is required — but only occasionally
- High-cardinality columns dominate model size
- Historical detail is rarely queried
- Drill-through needs detail that summary reporting does not
- Most queries can be answered from an aggregated layer
- The analytical value of the detail does not justify its in-memory cost

Keep the common path fast. Isolate the expensive path.



Interactive challenge #2

- You've just seen these techniques applied to Yellow Taxi. Now let's see if you can spot the same opportunities in a completely different model.
- What would you
  - Remove?
  - Aggregate?
  - or isolate?

Recap: Make the model smaller

Diagnose first
- Find the biggest contributors before changing anything.

1

Remove what you don't need
- Fewer rows, columns, and technical attributes.

2

Reduce cardinality
- Especially IDs, timestamps, precision, and free-text detail.

3

Remove unnecessary overhead
- Hierarchies, metadata, and features that add memory without value.

4

Isolate expensive detail
- Aggregate, move, or limit detail that is needed only occasionally.

5

Measure every change
- Validate the saving and the functional trade-off.

Optimize the biggest contributors first – then measure what you gained.

A smaller model can still fail during refresh

BREAK

The model fits. The refresh doesn't.

Make the Model Process Reliably

Diagnose Processing & Peak Memory

Diagnose Processing & Peak Memory

1. Why full refresh doubles memory
2. Steady state vs processing peak

VertiPaq has segments

Units of compression

Units of processing

Units of query paralellism

Tabular Editor

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.

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.

So, VertiPaq has segments

- Subset of a column's values (fixed number of rows)
- Partitions have >=1 segments

Tabular Editor

Under (RAM) 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

Units of parallelism

Tabular Editor

Segments

Units of compression

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

Tabular Editor

Tabular has partitions

- Again, VertiPaq has segments
- Partitions are a unit of operational management
  - Separate logical query for each partition
  - Separate storage mode per partition
  - What you manage with XMLA commands and APIs

- 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

Things you can do with

Segments

– Look at them

Things you can do with

Segments

- Look at them
- Set maximum size per segment with *DefaultSegmentRowCount*
  - 8 million default: SSAS, AAS, Large storage format PPU/Premium/Fabric
  - 1 million default: Power BI Desktop, Pro, small storage format
  - Tunable available in: SSAS, AAS

Things you can do with

Segments

- Look at them
- Set maximum size per segment with *DefaultSegmentRowCount*
  - 8 million default: SSAS, AAS, Large storage format PPU/Premium/Fabric
  - 1 million default: Power BI Desktop, Pro, small storage format
  - Tunable available in: SSAS, AAS

Partitions

- Edit
  - Create
  - Update
  - Delete
- Operate
  - Process (refresh)

Things you can do with

Segments

- Look at them
- Set maximum size per segment with *DefaultSegmentRowCount*
  - 8 million default: SSAS, AAS, Large storage format PPU/Premium/Fabric
  - 1 million default: Power BI Desktop, Pro, small storage format
  - Tunable available in: SSAS, AAS

Partitions

- Edit
  - Create
  - Update
  - Delete
- Operate
  - Process (refresh)
- Mess up segments: too-narrow filter predicates yield more segments (no segment may span partition bounds)

Refresh timeline and memory

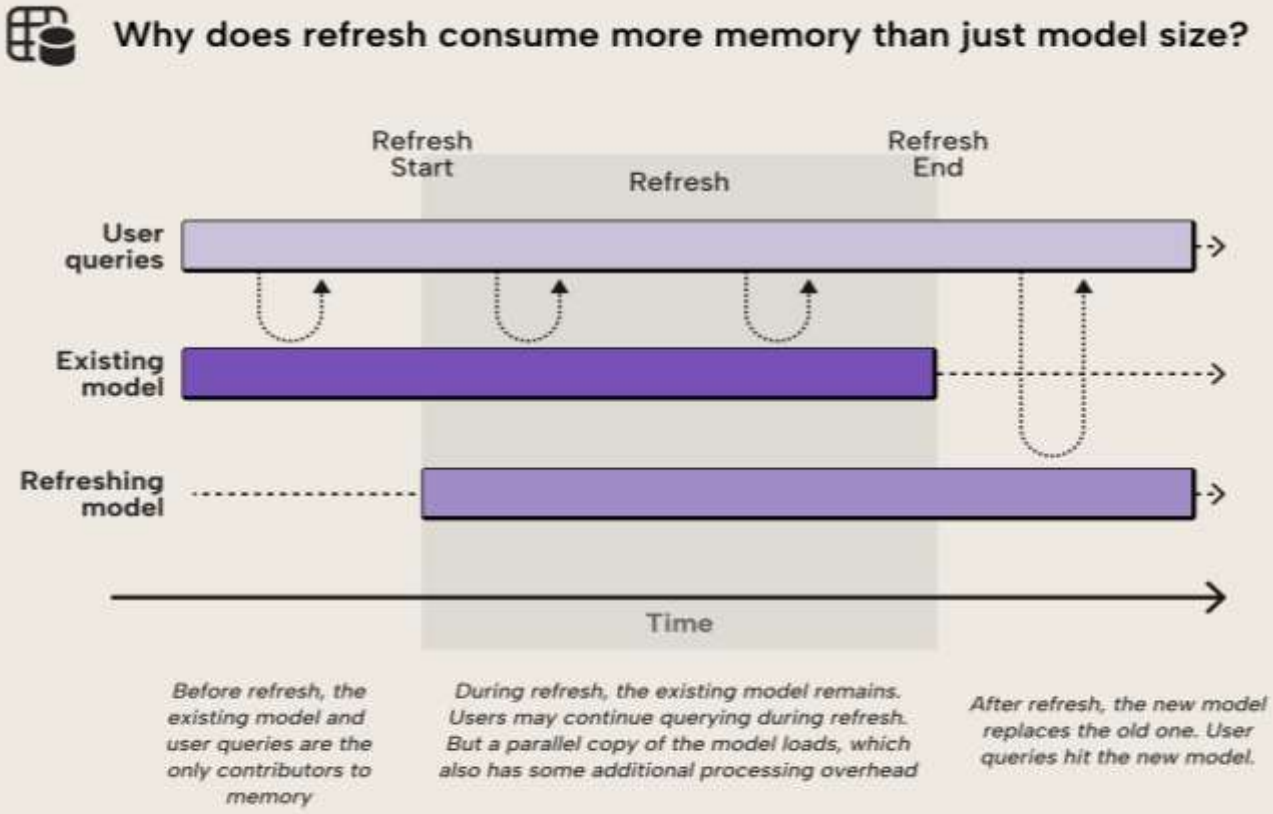
- Refreshing requires that we create new versions of all partitions being refreshed

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

Refresh timeline and memory

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

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



Tabular Editor

Partition strategies

Tabular Editor

Partition strategies

1. Partitioning
2. Incremental refresh
3. Custom partitions
4. Hybrid tables
5. Choose your partition strategy

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

Storage modes

Import

- Most stable
- Full DAX and TOM feature coverage
- Best performance in most scenarios
- Bound by RAM
- Requires refresh orchestration

DirectQuery

Direct Lake

Storage modes

Import

- Most stable
- Full DAX and TOM feature coverage
- Best performance in most scenarios
- Bound by RAM
- Requires refresh orchestration

DirectQuery

- DAX->SQL
    - With limitations
- Worse in nearly every performance metric than alternatives
- Bound by back-end storage+compute
- Every query sees latest data in back-end

Direct Lake

(does not have a space in its name)

Tabular Editor

Storage modes

Import

- Most stable
- Full DAX and TOM feature coverage
- Best performance in most scenarios
- Bound by RAM
- Requires refresh orchestration

DirectQuery

- DAX->SQL
  - With limitations
- Worse in nearly every performance metric than alternatives
- Bound by back-end storage+compute
- Every query sees latest data in back-end

(does not have a space in its name)

Direct Lake

- Parquet -> VertiPaq with no optimization
- Bound by RAM
- Amortizes refresh + data processing across many queries
- Most complex orchestration and operational management

(does have a space in its name)

Tabular Editor

Incremental Refresh

- Power BI feature
- Policy-based on time ranges
- Automatically drops too-old partitions
- All import
- Good enough for many scenarios

- Partition queries **must** be evaluable against source without reading all data
    - Usually this means folding to SQL

Custom partitions

- Create partitions with a tool other than PBI Desktop (Tabular Editor 2 or 3 or CLI, Visual Studio, editing TMSL/TMDL, XMLA commands to create partitions)
- Any query separation you would like
- Requires your own orchestration
- Gives much more control, e.g.:
  - Different back-end for each partition
  - Any retention policy for old partitions

Hybrid tables

- Power BI feature:
  - Incremental refresh for old data
  - DirectQuery for the latest partition
- Custom
  - Any partition can be DQ and any can be import
  - E.g., DirectQuery for archival data, and import for latest data



Interactive discussion #3

- How would you partition; what would you want to know about the data for
  - Decades of history



Interactive discussion #3

- How would you partition; what would you want to know about the data for
  - Decades of history
  - Immutable previous months, daily data current month



Interactive discussion #3

- How would you partition; what would you want to know about the data for
  - Decades of history
  - Immutable previous months, daily data current month
  - Immutable previous months, streaming data for current month



Interactive discussion #3

- How would you partition; what would you want to know about the data for
  - Decades of history
  - Immutable previous months, daily data current month
  - Immutable previous months, streaming data for current month
  - Maximum refresh window: 20 minutes

Recap

- Incremental refresh → predictable time-based history
- Custom partitions → full control
- Hybrid tables → combine Import and DirectQuery
- Partition size affects both processing and maintenance
- More partitions are not automatically better

Direct Lake

Direct Lake

1. Direct Lake changes where the work happens
2. Transcoding and reframing
3. Row groups, V-Order, OPTIMIZE
4. Direct Lake is not a free lunch

Direct Lake

- Loads data into VertiPaq engine in-memory

Tabular Editor

Direct Lake

– Loads data into VertiPaq engine in-memory

Tabular Editor

Direct Lake

- Loads data into VertiPaq engine in-memory

Tabular Editor

Direct Lake

–Loads data into VertiPaq engine in-memory

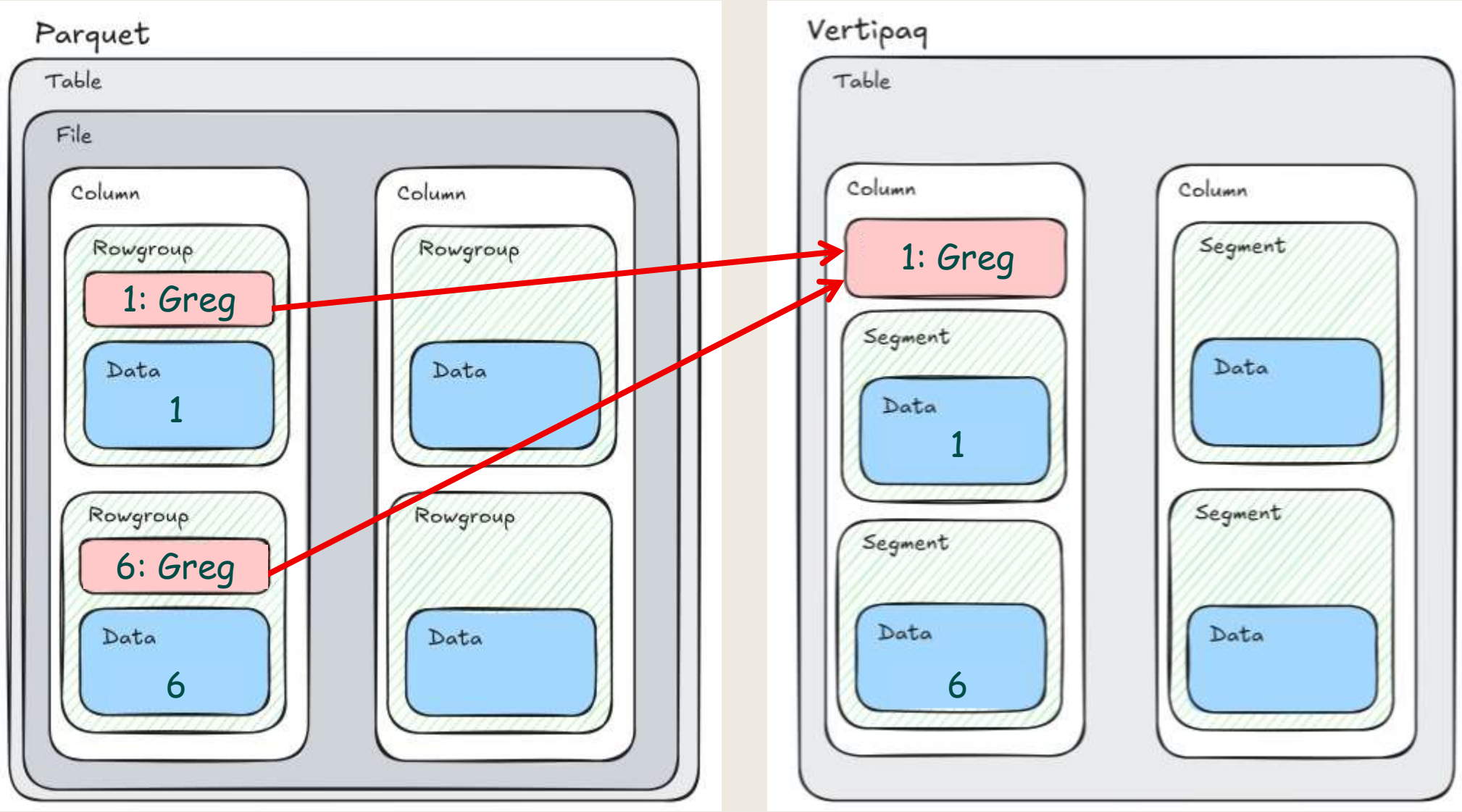
Tabular Editor

Transcoding

- Loads data into the VertiPaq engine in-memory
- Name for the 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

Tabular Editor

Direct Lake - Transcoding



Tabular Editor

Reframing - Swapping in and out of memory

- The word for refreshing a Direct Lake model: pointing it to a new snapshot
    - Evicts updated columns from memory
    - Clears VertiPaq caches
- It's nothing new. Dataset evictions and dynamic memory management is a core concept when we moved to the architecture behind Power BI Premium Gen2
- Only works on the exact schema from Direct Lake: anything you might do in Power Query for an Import model must be done elsewhere

Direct Lake changes where the work happens

Import	Direct Lake	What changed?
Refresh loads data into VertiPaq	Data is transcoded on demand	Processing shifts from refresh time towards query/runtime
Power Query can shape the data	Model expects lake schema	Transformations move upstream
VertiPaq decides compression/sort behavior	Row groups come from Parquet/Delta	Upstream layout matters more
Refresh replaces model data	Reframing points to a new snapshot	Refresh semantics change
Model optimization happens mostly in the semantic layer	Optimization spans lake + semantic model	Responsibility moves upstream

Direct Lake does not remove optimization – it moves part of it upstream.

Tabular Editor

Row groups, V-Order, OPTIMIZE

- 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 OPTIMIZE and table maintenance

Direct Lake is not a free lunch

- Generally: a way to shift compute and refresh resources around
- Not magic
- Testing benefits are 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**

Direct Lake is not a free lunch

What you gain

- No traditional Import refresh
- Faster data availability through reframing
- Less refresh orchestration
- Potentially lower processing pressure

What you still pay for

- Semantic model memory
- Transcoding
- Poor file and row-group layout
- OPTIMIZE and table maintenance
- Upstream engineering complexity

You’re shifting cost – not eliminating it.

Tabular Editor

Refresh Scale-Out

Tabular Editor

Refresh Scale-Out

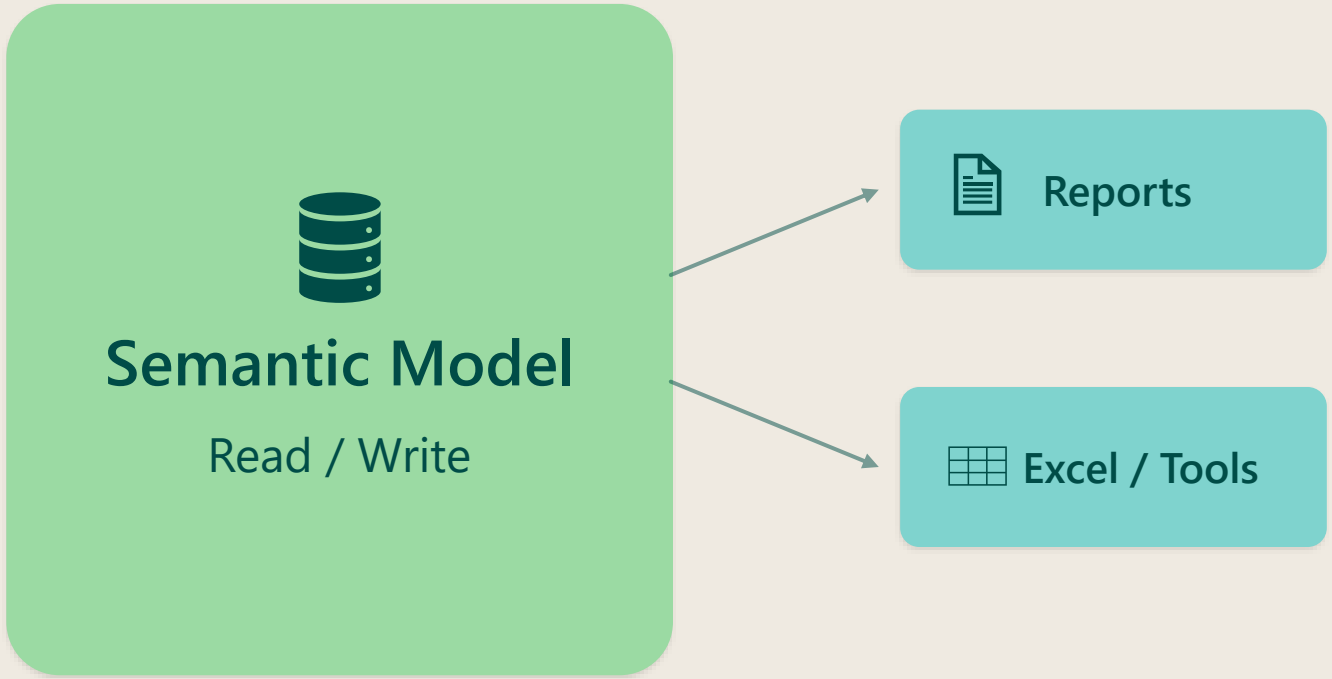
1. The Hidden Gem
2. How scale-out works
3. When to use it
4. What problem it solves
5. Why it’s not magic

Refresh Scale-Out

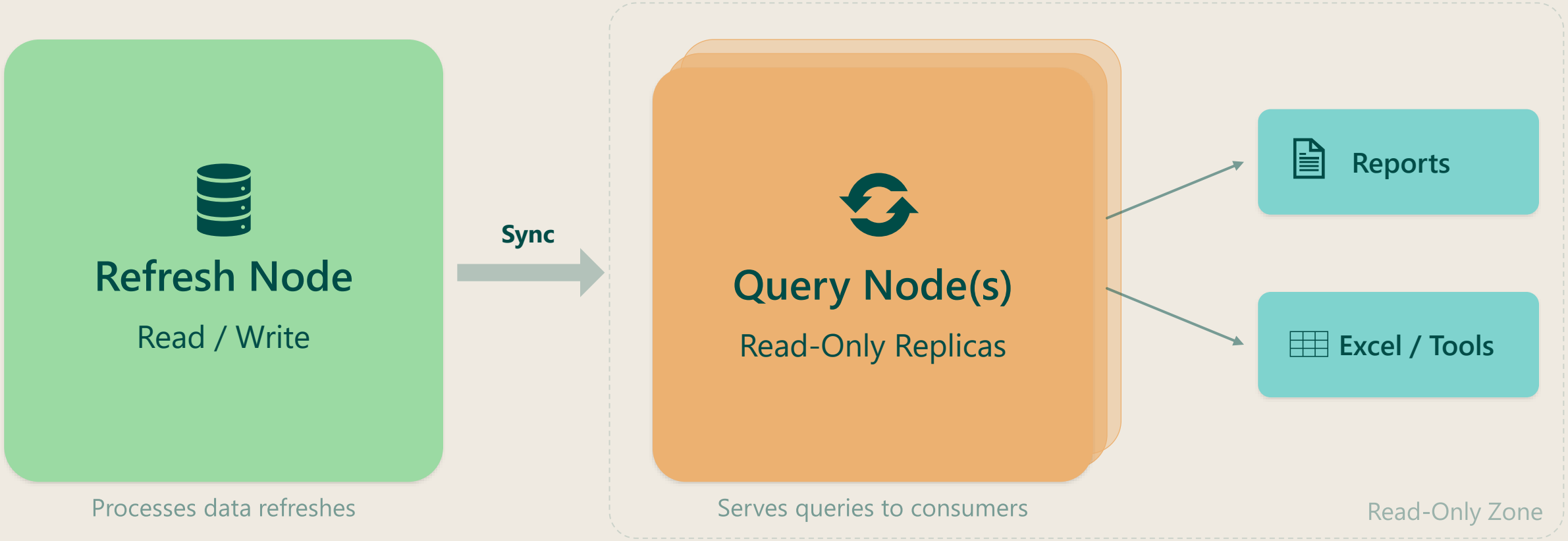
- Officially known as Query scale-out, and has been available in Azure Analysis Services for as long as I can remember
- Consists of a processing node and multiple read-only query nodes, and is a feature for Semantic Models that are consumed by a large audience
- Can also be used to optimize memory while 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.*

Normal Architecture



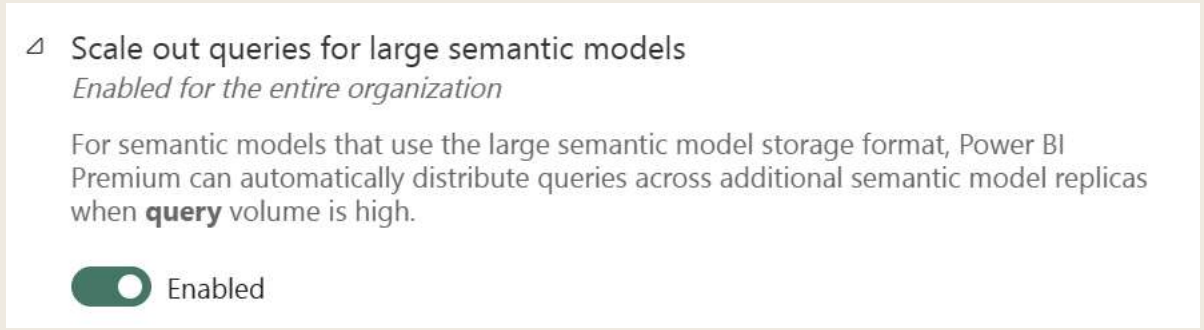
Refresh Scale-Out Architecture



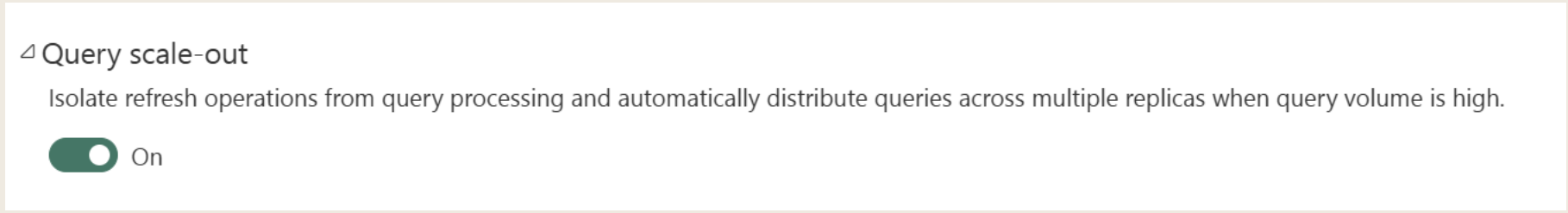
Tabular Editor

Configure Refresh Scale-Out

- By default, it is enabled on the tenant level

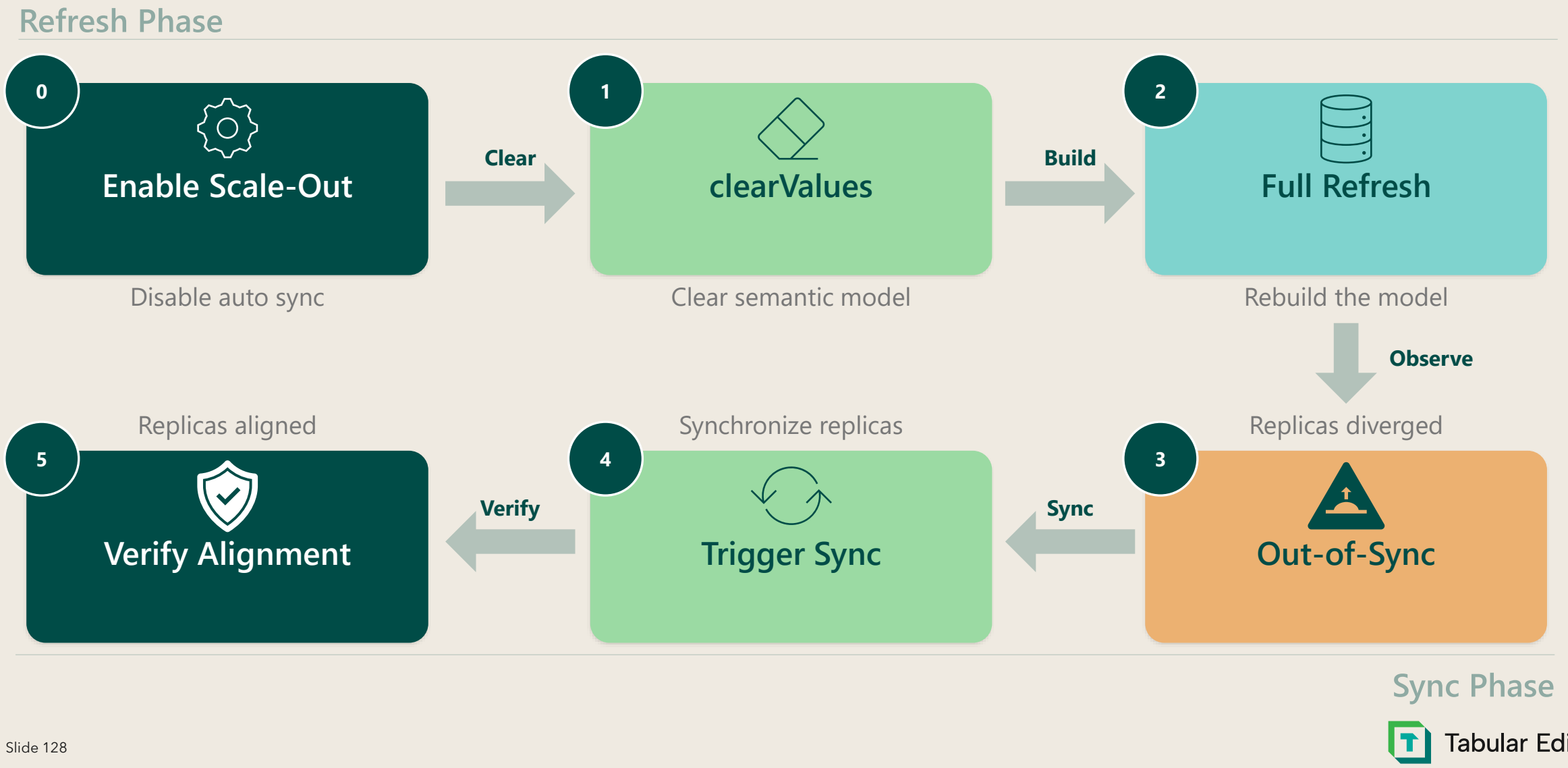


- Change the “maxReadOnlyReplicas” on the semantic model level from the default value of 0



Tabular Editor

Refresh Scale-Out Process



Tabular Editor

When scale-out helps, and when it doesn’t

Good fit

- Heavy report/query concurrency
- Refresh and user queries compete for resources
- Processing needs to be isolated from consumers
- Large refreshes approach the model memory limit
- You can control when replicas synchronize

Not a fix for

- An oversized semantic model
- Inefficient partition or refresh design
- Slow DAX or poor model design
- Low query concurrency
- Avoiding proper capacity planning

Optimize first. Scale second.

Make the Model Query Efficiently

Diagnose query performance

Tabular Editor

Diagnose query performance

1. Where does query time go?
2. Storage Engine vs Formula Engine
3. Diagnosing a slow query
4. DEMO: DAX Studio

Where does query time go (when it's not around here)?

Engines

- Storage Engine: very parallel, very fast, very primitive
  - Data scanning: direct operations on segments
  - Relationship propagation
  - Basic aggregations
- Formula engine: single threaded
  - Advanced logic, conditionals

Specific things that can happen

- Callbacks: SE query that depends on FE results
- Materialization: intermediate results stored in RAM to compute a later result
- Virtual relationships: skipping the VertiPaq relationship index

VertiPaq has segments

Units of compression

Units of processing

Units of query paralellism

Tabular Editor

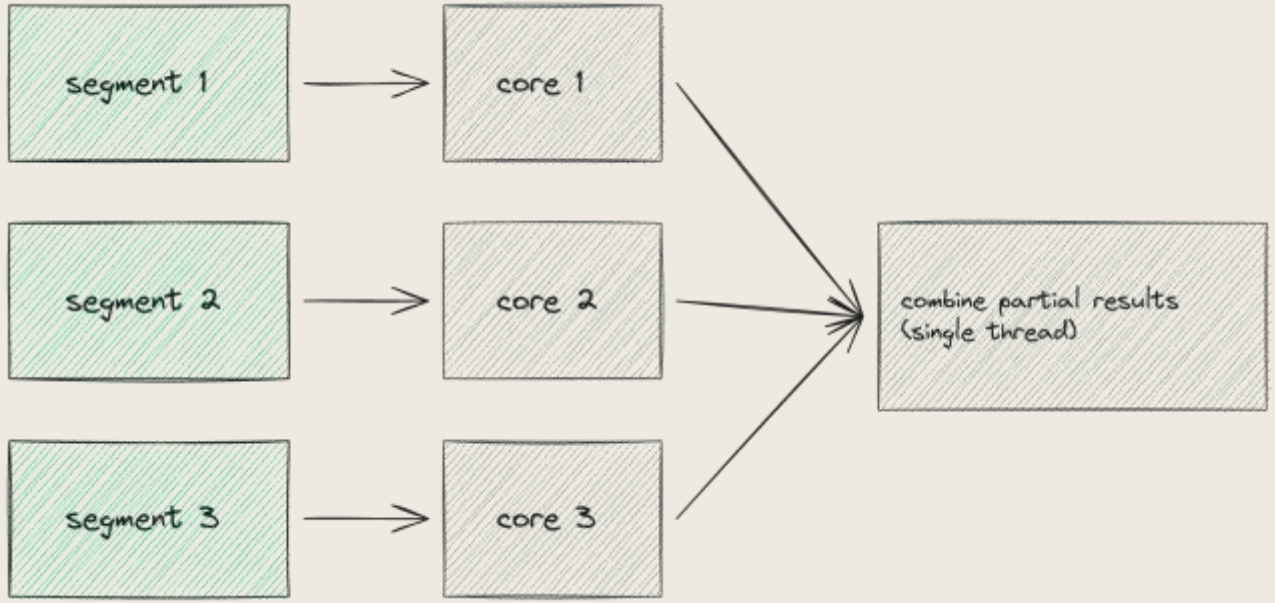
VertiPaq has segments

Units of query paralellism

- VertiPaq queries 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 Editor

VertiPaq has segments



Units of query paralellism

- VertiPaq queries 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 Editor

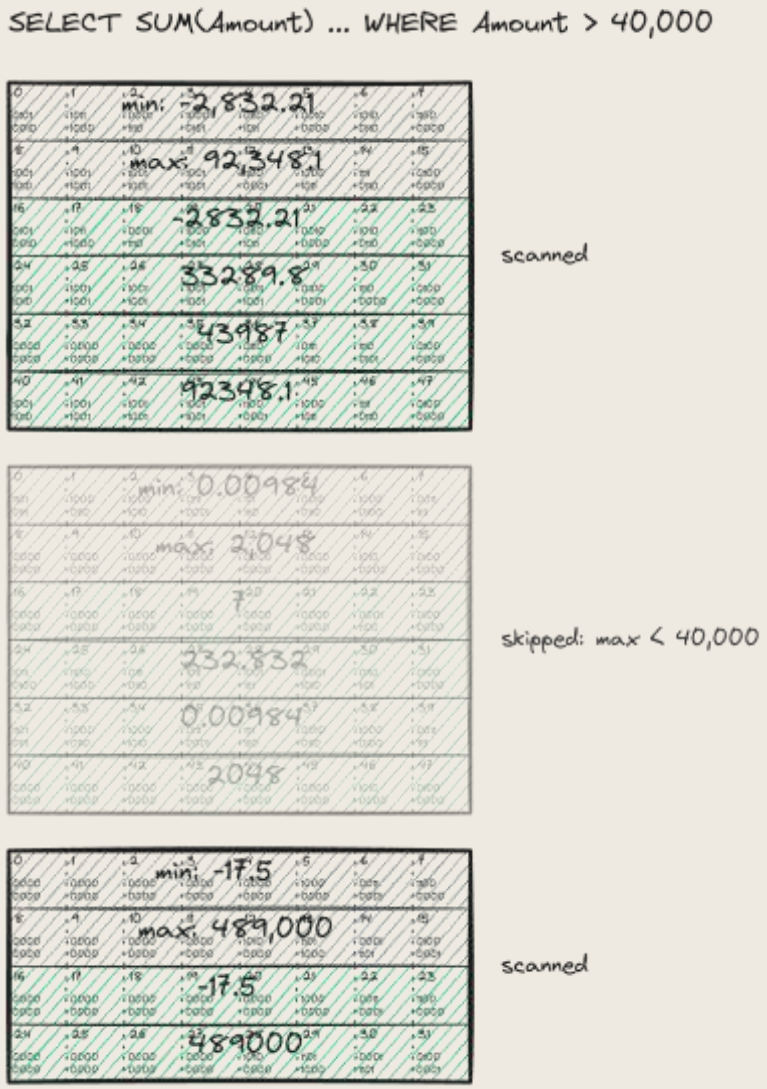
Segment elimination

- Similar to partition elimination in RDBMSes
    1. Compare segment metadata to query filters
    2. Identify segments holding no useful data
    3. Skip them

- Another benefit of the sorting that VertiPaq does

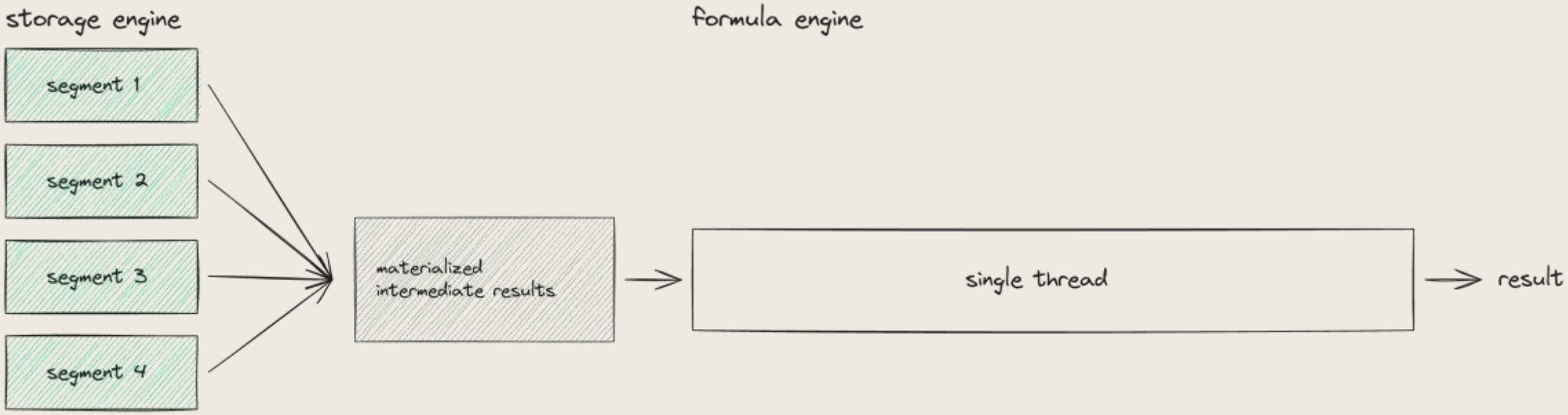
Segment elimination

- Similar to partition elimination in RDBMSes
    1. Compare segment metadata to query filters
    2. Identify segments holding no useful data
    3. Skip them
- Another benefit of the sorting that VertiPaq does



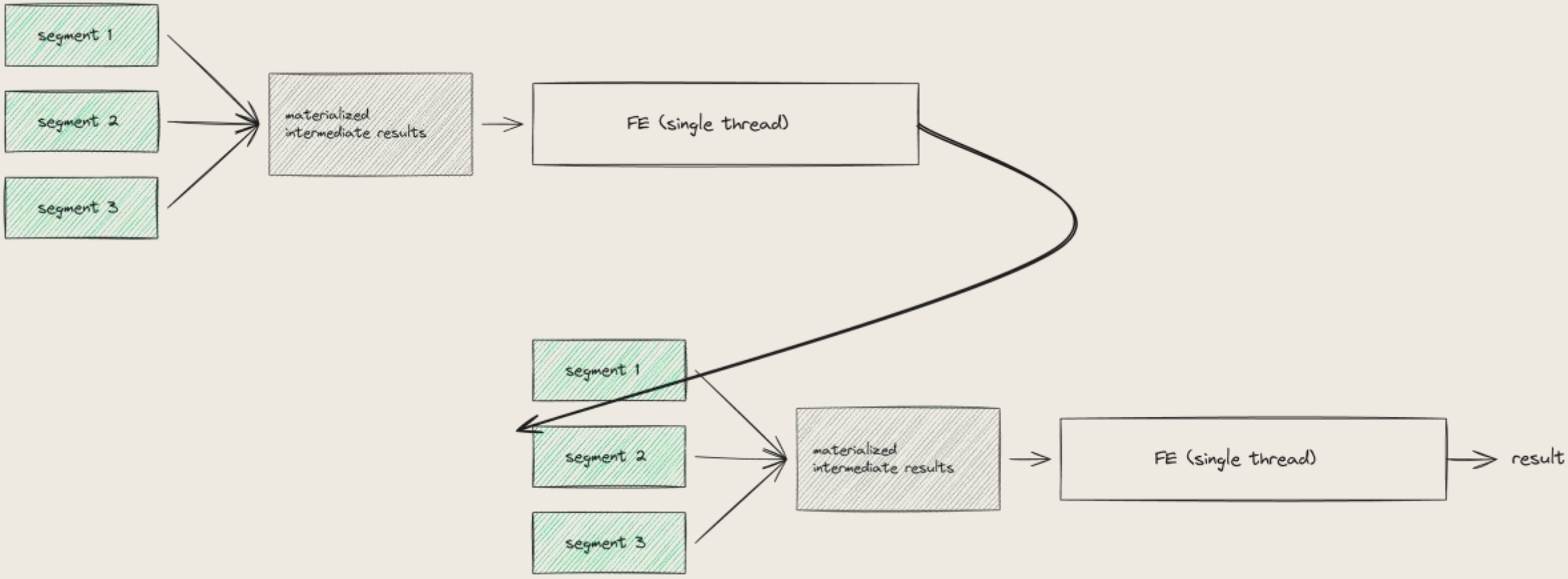
Tabular Editor

Basic query flow



Tabular Editor

Callback data flow



Tabular Editor

Relationship problems?

fact (logical)

DimKey
8
7
7
5
1
3
8
6
7
8

dim (logical)

DimKey	Name
3	Name3
7	Name7
1	Name1
5	Name5
8	Name8
2	Name2
6	Name6
4	Name4

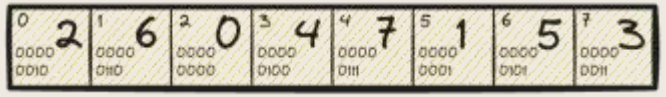
dim: DimKey dictionary

DictKey	DimKey
0	1
1	2
2	3
3	4
4	5
5	6
6	7
7	8

fact: DimKey column page



dim: DimKey column (encoded)



Tabular Editor

Relationship problems?

fact (logical)

DimKey
8
7
7
5
1
3
8
6
7
8



fact: DimKey column page

0 8 0000 1000	1 7 0000 0111	2 7 0000 0111	3 5 0000 0101	4 1 0000 0001	5 3 0000 0011	6 8 0000 1000	7 6 0000 0110
8 7 0000 0111	9 8 0000 1000	10 0000 0000	11 0000 0000	12 0000 0000	13 0000 0000	14 0000 0000	15 0000 0000

dim (logical)

DimKey	Name
3	Name3
7	Name7
1	Name1
5	Name5
8	Name8
2	Name2
6	Name6
4	Name4

dim: DimKey dictionary

DictKey	DimKey
0	1
1	2
2	3
3	4
4	5
5	6
6	7
7	8

dim: DimKey column (encoded)

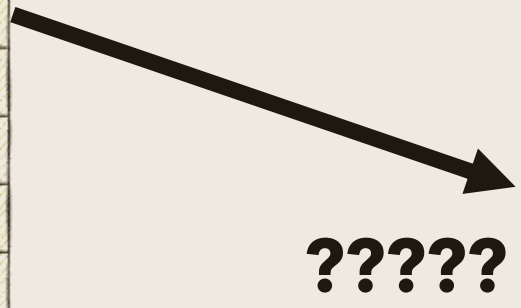
0 2 0000 0010	1 6 0000 0110	2 0 0000 0000	3 4 0000 0100	4 7 0000 0111	5 1 0000 0001	6 5 0000 0101	7 3 0000 0011

Tabular Editor

Relationship problems?

fact (logical)

DimKey
8
7
7
5
1
3
8
6
7
8



?????

Different encodings and keys on different sides of the relationship

dim (logical)

DimKey	Name
3	Name3
7	Name7
1	Name1
5	Name5
8	Name8
2	Name2
6	Name6
4	Name4

dim: DimKey dictionary

DictKey	DimKey
0	1
1	2
2	3
3	4
4	5
5	6
6	7
7	8

fact: DimKey column page

0 8 0000 1000	1 7 0000 0111	2 7 0000 0111	3 5 0000 0101	4 1 0000 0001	5 3 0000 0011	6 8 0000 1000	7 6 0000 0110
8 7 0000 0111	9 8 0000 1000	10 0000 0000	11 0000 0000	12 0000 0000	13 0000 0000	14 0000 0000	15 0000 0000

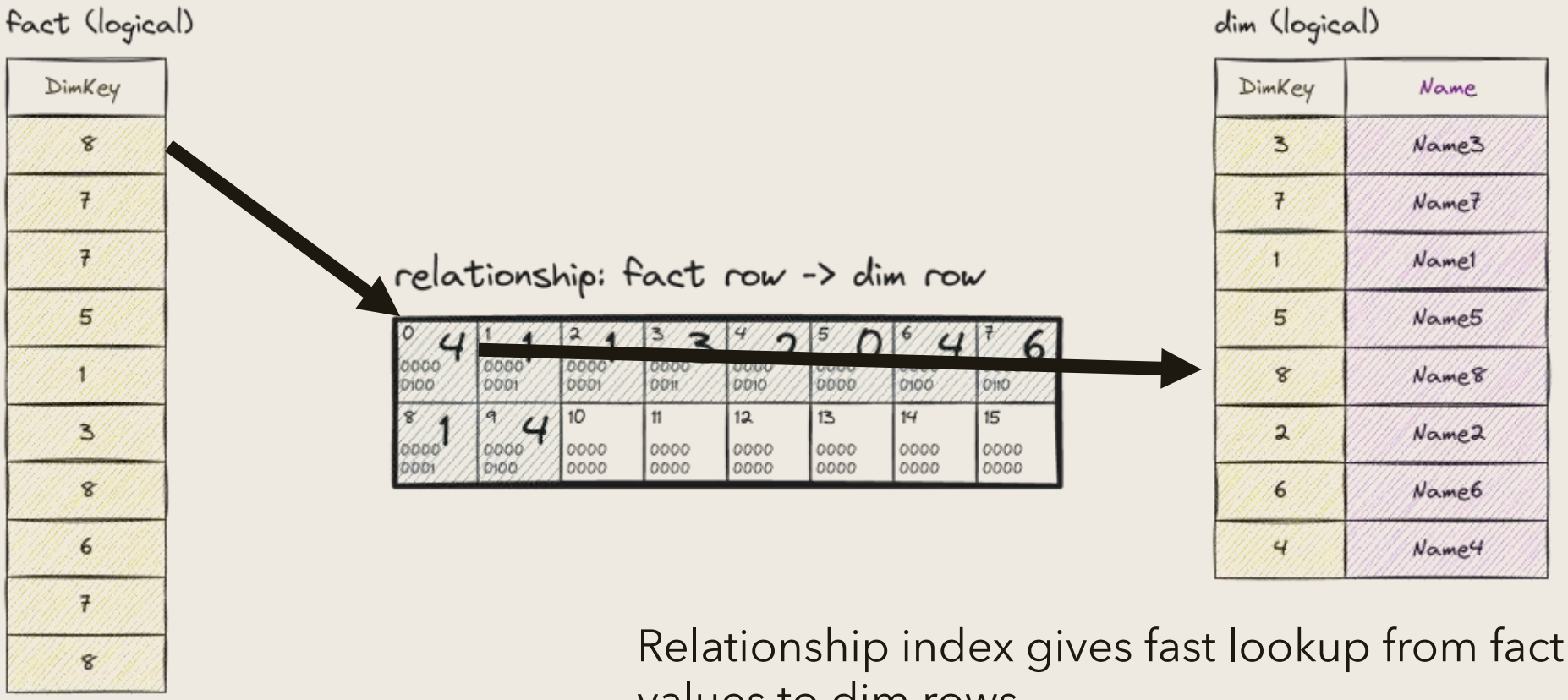
dim: DimKey column (encoded)

0 2 0000 0010	1 6 0000 0110	2 0 0000 0000	3 4 0000 0100	4 7 0000 0111	5 1 0000 0001	6 5 0000 0101	7 3 0000 0011

Tabular Editor

Relationships take work

(also life advice)





Relationship index gives fast lookup from fact values to dim rows.

Don't ignore your relationships or go outside of them (<<also life advice)

Tabular Editor

Model design and query performance

Tabular Editor

Model design and query performance

1. Model design affects query performance
2. Relationships are not free
3. DAX patterns that hurt

Tabular Editor

Wrap-up

Problem	Technique	Trade-off
Model too large	Reduce columns/cardinality	Less detail
Detail too expensive	Aggregations / isolation	Complexity / DQ latency
Refresh fails	Partitions / incremental	Orchestration
Refresh competes with users	Scale-out	Complexity / capacity
Direct Lake performs poorly	Optimize upstream layout	Data engineering work
Queries are slow	Model + DAX optimization	Development effort

End-to-End Optimization Steps

Step 1: Diagnose

- What is consuming memory?
- What causes refresh peak?
- Which queries are slow?
- What should we fix first?

Step 2: Reduce

- Remove unused columns
- Reduce cardinality
- Disable unnecessary hierarchies
- Isolate expensive detail
- Re-measure

Step 3: Process smarter

- Partition
- Incremental refresh
- Consider hybrid / Direct Lake
- Reduce peak processing memory
- Re-measure

Step 4: Query smarter

- Inspect Server Timings
- Simplify relationships where possible
- Optimize expensive DAX
- Re-measure

Performance is not something you fix at the end. It's something you design for.

Tabular Editor

Resources

Slides

- Optimizing Semantic Model Memory in Fabric (tabulareditor.com/blog)
- Model Performance Tuning in DAX Studio with Marco Russo (daxstudio.org/docs/videos)
- Building a Fabric Lab for Semantic Model Memory Experiments (justb.dk/blog)

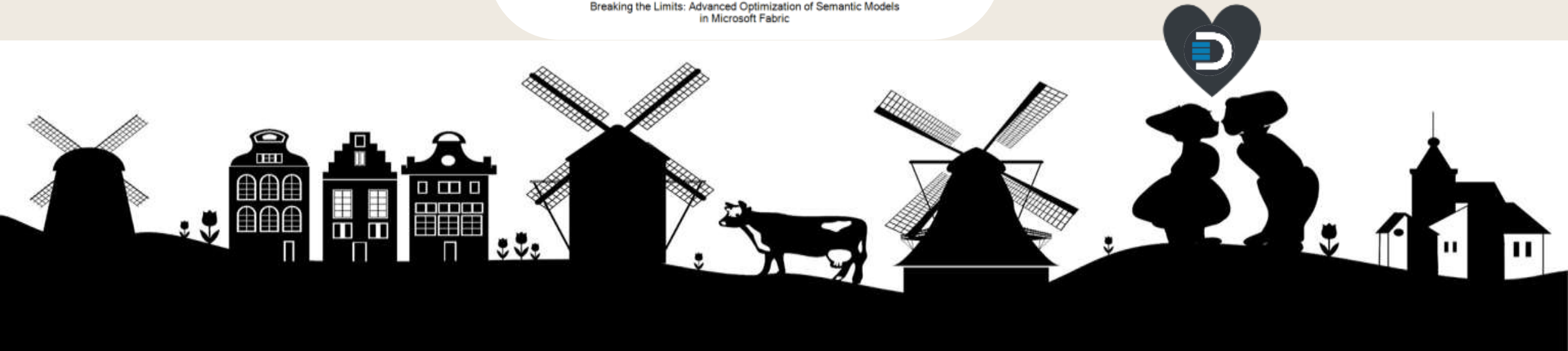
Loved it? Learned something? Tell us! Share your feedback in just 1 minute



Tabular Editor

Rate This session

Breaking the Limits: Advanced Optimization of Semantic Models in Microsoft Fabric



Tabular Editor



Thank you for your time!

linkedin.com/in/gregbaldini/

linkedin.com/in/blindbaek/

Thank you to the sponsors!

























