Modeling for AS Tabular scalability

Microsoft BI Technical Article

Writer: Daniel Rubiolo.

Contributors and Technical Reviewers: Akshai Mirchandani, Christian Wade.

Published: August 2018.

Applies to: Microsoft SQL Server 2017 Analysis Services, Microsoft Azure Analysis Services.

Summary: This whitepaper and associated samples describe design techniques for modeling Tabular projects to optimize scalability.

Contents

Introduction

In scenarios where we have models with very big volumes of data (tables with multi-billion rows), we should consider some data preparation and modeling best practices so that we help the engine do its best work in compressing it. This would maximize getting as much of the data as possible into memory, so users can consume it in their analyses.

AS (Analysis Services) at its core has an in-memory columnar database engine (referenced here as Tabular, a.k.a. VertiPaq), which during data processing converts the rows of data from the source into an optimized storage organized by columns, encoding & compressing each column's data individually. This compressed data allows the engine to fit more of it into memory and speeds up the vast majority of analytical queries. Depending on the query scenarios/requirements, we will reach a balance point between compression (get more into memory; read less) and decompression at query time (more CPU processing).

In this article we cover a simplified overview of how the Tabular engine works, and some data design best practices that take the most advantage of the engine's inner workings. We cover a real-world example to discuss these practices in action, in which we obtained ~84% compression, allowing 5 times more data in the same capacity.

> ***Note 1****: this article directly applies to SQL Server AS (SSAS) and Azure AS (AAS), even though the example we cover uses AAS. The scalability gains from the design techniques discussed in this article (not the hints & segment's configuration) also apply to models in Excel's PowerPivot and Power BI (Desktop and the Service), as they all share an equivalent in-memory Tabular engine.*
>
> ***Note 2****: we do not cover here topics such as partitioning, scaling out, and high availability; only data modeling techniques that assists Tabular with achieving better compression.*
>
> ***Note 3****: we assume in this article that you are already familiar with dimensional modeling, star schemas, and Kimball design techniques (we provide some links to resources in the references).*

Overview of how the Tabular Engine works

Usually in BI exploratory analytics, users only need a few columns at a time, getting aggregated values at high level, and adding details as they drill down. Tabular provides optimizations for these scenarios, by focusing on minimizing read time. On the other hand, if we query for many columns (such as in detailed reporting scenarios), Tabular will need more time to reorganize the data into rows again, consuming more CPU cycles and memory.

As mentioned earlier, Tabular encodes and compresses the data column by column during processing/refresh. Values in each column will have the same data type, so Tabular can optimize for this. It works in two stages:

1. It encodes the data into an integer value, either using value encoding or dictionary encoding:
    a. **Value encoding**: applied to numeric columns. It consists of using characteristics of the distribution of values in the column to reduce the number of bits needed to represent them (applying several mathematical techniques).
    b. **Dictionary encoding (a.k.a. Hash encoding)**: creates a list of distinct values in the column, indexes each value in the list with integers, and then replaces the values in the original column with these integer indexes.
2. Then it compresses the integer values, using different compression techniques (including both RLE and bitpacking compression algorithms):
    a. **RLE: Run Length Encoding:** it compresses data by removing consecutive duplicates in the column (keeping the count of the run), resulting in a more compact format, which benefits from the sort order of the data.
    b. **Bitpacking:** following the data compression's principle of "don't store bits of data unless absolutely necessary", this technique use algorithms to represent data with as few as possible number of bits.

Whichever the encoding that Tabular chooses (value or dictionary), the internal data structure will end up consisting of integers. At the start of the initial processing, Tabular uses a sample of data to determine the encoding. However, if during processing it determines the initial option chosen was wrong, it will re-encode. This could result on more processing time, so for big datasets it can be very beneficial if we provide good samples in the first partition.

Next, Tabular will consume some time trying to find the best sort order to maximize compression by analyzing the data. However, for big datasets this can prove very time consuming, so if we use our knowledge of the data to provide it already ordered, then we can maximize the chances of getting the best compression.

After the previous steps, Tabular will then create calculated columns and compress them. Because this happens after compression, these calculated columns are not considered for the sort order, which opens an opportunity to drive more efficiency. If we can make these calculations at the source, we may be able to drive even better compression by having the chance of using calculated columns as part of the sort order evaluation.

Finally, Tabular builds the data structures for relationships and hierarchies. It uses hierarchies to optimize MDX queries, which means that we can save more space by disabling some unneeded hierarchies (explained shortly). In the case of relationships, Tabular builds data structures mapping keys in one table to row numbers in another, so that at query time when given some keys (i.e. by filtering), it can easily find the related records, making slicing & dicing analytical activities very fast (a core feature optimized for star schemas).

To optimize data processing, Tabular processes the data in segments of 8 million rows (by default). It reads a segment and starts compressing it while reading the next one. It also uses the segments at query time to read in parallel across cores. Compression will happen by segment. With larger segments, Tabular obtains better compression because it can use more data to compress, but it'll take more time. With smaller segments, Tabular will achieve more parallelism, but may take longer in consolidating results and thus take more query time. We can try different segment sizes to achieve optimal compression in very large tables. We also must consider that segments cannot exceed the partition size, so if we use partitioning, we must coordinate this dependency (using small partitions is a very common mistake).

> ***Note 4****: The data in a model is "always available" by design, meaning that the Tabular engine allows the current data to be available for users' queries while the refresh happens in a transaction copy (also in memory) while compression happens. When the refresh in the transaction completes, it gets committed and processing memory gets released. For big datasets, we must plan for this additional memory processing requirement (which varies with full or partial refreshes).*

> ***Note 5****: Power BI Desktop and PowerPivot are optimized for smaller datasets, so their segment's default is 1 million rows and cannot be changed.*

Optimizing the data model for Tabular

For complex data models, especially with many inter-related big tables, we should consider "Dimensional Modeling" (a.k.a. "Star Schemas" or "Kimball modeling"). These designs produce optimized "dimension" and "fact" tables. Dimensions contain most of the text fields driving filtering, categorizations, grouping, and labels around unique keys; while Facts contain (for the most part) a collection of pointers (foreign keys) to these dimensions, and the fields with numeric measures to produce metrics & KPIs (key performance indicators). Dimension tables are usually relatively small compared to Facts, which in turn are the ones potentially having multi-billion rows in big models (i.e. transactions; logs). In addition, if we provide all the keys as integers when they get into Tabular, then we could achieve the most compression and scale.

> ***Note 6****: At query time, Tabular will use the storage engine to retrieve the data needed, including navigating relationships. Using star schemas, the dimensions usually are the small tables in the model with the fields for displaying in charts and slicers that users will click to filter the measures coming from fact tables. So, the small tables will filter the big tables. This is very important for performance at query time (and also for compression, described below).*

Dimensional models have other important advantages for big models. They drive data simplicity, minimizing complexity for users in interpreting the meaning of the data model. They also enable advanced analytical scenarios such as keeping history of changes, empowering analysts to understand the evolution of the data over time. (See References for more.)

Steps for optimizing Dimensions:

1. Minimize the number of columns.
2. Reduce cardinality (data type conversions).
3. Filter out unused dimension values (unless a business scenario requires them).
4. Integer Surrogate Keys (SK).
5. Ordered by SK (to maximize Value encoding).
6. Hint for VALUE encoding on numeric columns.
7. Hint for disabling hierarchies on SKs.

Steps for optimizing Facts:

1. Handle early arriving facts. *[Facts without corresponding dimension records.]*
2. Replace dimension IDs with their surrogate keys.
3. Reduce cardinality (data type conversions).
4. Consider moving calculations to the source (to use in compression evaluations).
5. Ordered by less diverse SKs first (to maximize compression).
6. Increased Tabular sample size for deciding Encoding, by considering segments and partitions.
7. Hint for VALUE encoding on numeric columns.
8. Hint for disabling hierarchies.

Use case example scenario

I recently worked in a project in which the Services team delivered to us (the BI team) the data as text files in an Azure Data Lake, as result of processing logs thru a Spark-based solution. Following some design best practices already, they processed the raw data from the services' logs into a star schema (dimensions and facts) to minimize data redundancies, adding very valuable data quality and reference integrity in the process. The first fact table in our model was close to ½ billon rows for 3 months of data.

Nothing that AAS couldn't handle… for now.

However, we expected to have details for at least a year, and historical for 3 years, so the data was expected to grow to much more as part of processing additional time periods and added more metrics with additional fact tables.

The main challenge we had was that the star schema as provided was using the business keys, which were all GUIDs (long strings). The services team didn't see the need for surrogate keys (one of the foundational design principles on any data warehouse) because we didn't need to keep change history… yet. However, there are other very important uses for surrogate keys (SKs), and optimizing compression for high scalability is one of them. Another reason they didn't provide SKs was the difficulty for their big data tech to provide unique values across processing nodes in an effective way.

> ***Corollary***: *"Not just because you can, you should". There are appropriate ways to use features. Get used to design with best practices in mind, so you maximize the capabilities of your models.*

Initial assessment

To simplify the scenario, the initial model had:

- A fact-less fact table with ½ billon rows, linked to 4 dimensions (the facts modeled the relationships across dimensions).
- The biggest dimension had ~48 million unique records.
- The other 3 dimensions were small (3k records; 14 records; 3 records).

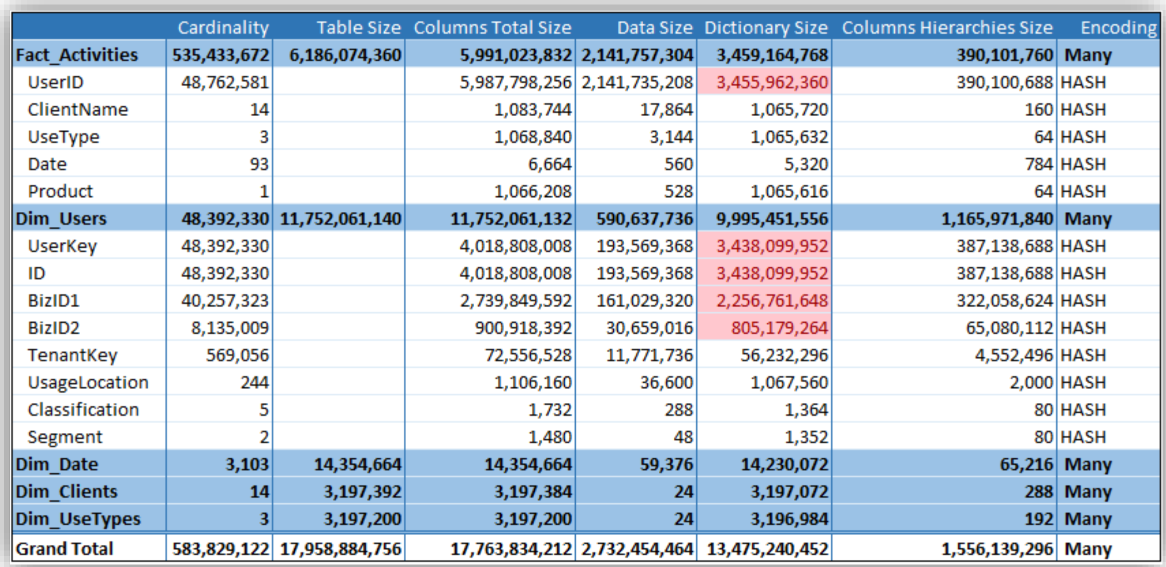
However, all the keys in these dimensions were text GUIDs (or other text keys), which were also used across all fact tables relating to these dimensions. Initially, they automated the delivery of the data from the lake's TSVs into AAS. The data could get into current sizes of AAS, but we expected we would run out of memory very soon.

As Tabular processed these data, the following would occur with all these GUIDs:

- Encode the dimension's unique key's GUID using Dictionary Encoding, converting them into integers.
- Encode the foreign key's GUID referencing the dimension also in the Fact table using Dictionary Encoding. This implies that with this model, each Fact table will have another Dictionary for that same column, mapping the unique values to another set of integers.
- Create the relationship data structure between the two.

If these GUIDs were integers instead when they get into AAS, then no dictionaries would be needed and most probably Value encoding could be used, resulting in better compression and leaving much more memory free for more data.

Using "VertiPaq Analyzer" (see links in References), this was the initial resource consumption of the data set in AAS:



	Cardinality	Table Size	Columns Total Size	Data Size	Dictionary Size	Columns Hierarchies Size	Encoding
Fact_Activities	**535,433,672**	**6,186,074,360**	**5,991,023,832**	**2,141,757,304**	**3,459,164,768**	**390,101,760**	**Many**
UserID	48,762,581		5,987,798,256	2,141,735,208	3,455,962,360	390,100,688	HASH
ClientName	14		1,083,744	17,864	1,065,720	160	HASH
UseType	3		1,068,840	3,144	1,065,632	64	HASH
Date	93		6,664	560	5,320	784	HASH
Product	1		1,066,208	528	1,065,616	64	HASH
Dim_Users	**48,392,330**	**11,752,061,140**	**11,752,061,132**	**590,637,736**	**9,995,451,556**	**1,165,971,840**	**Many**
UserKey	48,392,330		4,018,808,008	193,569,368	3,438,099,952	387,138,688	HASH
ID	48,392,330		4,018,808,008	193,569,368	3,438,099,952	387,138,688	HASH
BizID1	40,257,323		2,739,849,592	161,029,320	2,256,761,648	322,058,624	HASH
BizID2	8,135,009		900,918,392	30,659,016	805,179,264	65,080,112	HASH
TenantKey	569,056		72,556,528	11,771,736	56,232,296	4,552,496	HASH
UsageLocation	244		1,106,160	36,600	1,067,560	2,000	HASH
Classification	5		1,732	288	1,364	80	HASH
Segment	2		1,480	48	1,352	80	HASH
Dim_Date	**3,103**	**14,354,664**	**14,354,664**	**59,376**	**14,230,072**	**65,216**	**Many**
Dim_Clients	**14**	**3,197,392**	**3,197,384**	**24**	**3,197,072**	**288**	**Many**
Dim_UseTypes	**3**	**3,197,200**	**3,197,200**	**24**	**3,196,984**	**192**	**Many**
Grand Total	**583,829,122**	**17,958,884,756**	**17,763,834,212**	**2,732,454,464**	**13,475,240,452**	**1,556,139,296**	**Many**

Some initial observations:

- Ordered by "data size" descending, so the big Fact table is first, and then the fields with the most consumption.
- As you can see, all fields including in this initial sample were using "HASH" (dictionary) encoding.
- The "dim_Users" dimension included 4 types of IDs ("UserKey" held the same value as "ID", which in turn was a selection of "BizID1" and "BizID2" depending on internal processing logic [names changed]). "TenantKey" was another GUID to a snowflake dimension (not included in this sample).
- Notice the "dictionary size" column, given the use of HASH encoding: ~13.5Gb of memory.
- Notice the "cardinality" column, which gives the number of unique values in the column/table (the fact having ~1/2 billion rows).

First step: Optimizing the model's design

The data in this case was already in a dimensional model. However, it lacked the surrogate keys (IDs as integers – See resources in the references). We decided to have an Azure SQL DW (ASDW) between the data lake and AAS, so we could drive the optimizations described in the following steps. With ASDW, getting the data from the TSVs in the lake was easy and fast using Polybase thru the External Tables feature. We then added identity columns to generate the surrogate keys into new tables, and then got the data in AAS from ASDW using a view that filtered the data set to only the columns/rows we needed. (Of course, we implemented additional logic to manage incremental data refreshes.)

Second step: Optimizing dimension tables

We implemented the following design best practices for optimizing Dimensions:

1. Minimize the number of columns:
    a. There were no use cases needing the GUID keys for analysis, so we didn't get them into AAS (only the surrogate key from step 4 below).
    b. The "Product" field in the Fact depends on "ClientName", so we replaced both with a foreign key to the dim_Clients dimension.
    c. In general, identify columns in your model that are not required for your users' analyses, and exclude them. In particular, columns with high number of distinct values (i.e. primary keys, which cannot be compressed much). This is particularly important in fact tables, where the most records will be, driving overall model size (i.e. TransactionID, TimeStamps); however, usually not that important in dimensions, which are very small in comparison.
2. Reduce cardinality (data type conversions).
    a. Here is where you would, for example, convert DateTimes to a Date, or reduce the precision of numeric fields, which results in better compression ratios as a consequence of reducing the number of unique values.
    b. For example, a measure with 3 decimals from source may be rounded to 1 or to integer if your users don't need this level of precision in their analyses.
    c. Another example, a datetime field with precision to the millisecond can be split in two fields (date and time) with the time rounded to the minute.

   d. In the case of DateTimes, ask yourself if the time part is really needed, and discard it if not. If needed, split the column into its Date & Time parts (less unique values on each column, thus much better compression). An additional optimization would be to convert these two fields into surrogate keys to Date and Time dimensions (specially in fact tables).

3. Filter out unused dimension values (unless a business scenario requires them).
   a. If you don't need the dimension records that are not referenced in the facts, then you can filter them out to save some space.
   b. For example, you can see that the Date dimension has 3k records, while only 92 are used in the fact, so you might want to filter down to only the 3 months used (this is a small example, but in bigger & more complex models this technique may lead to significant gains).
   c. Do not use this if there's user requirements to, for example, count the total number of days, or summarize the data by day including empties, or know which days don't have related facts, etc.
4. Integer Surrogate Keys (SK).
   a. After getting the dimension's data into ASDW using Polybase, we copied it into new tables having an "ID" column as LONG with IDENTITY, so ASDW generated the surrogate keys for us (see details in References).
   b. Getting the data into AAS, we filtered out all the GUID keys and got loaded only the surrogate key and the few other useful fields.
5. Ordered by SK (to maximize Value encoding).
   a. Some testing needed: should we order by SK first to favor Value encoding for the most distinct column, or by the least cardinality fields first, to make them smaller with compression? It will depend on the characteristics of your data. Spending time on testing alternatives based on your knowledge of your data may lead to significant gains.
   b. The sort order of the source data can influence how much effort it takes for Tabular to identify the best compression, and it may exceed the time limit that is set for the compression phase. It is possible to improve the compression ratio and the speed of the compression phase by pre-sorting the data. But be careful with this approach, because you may end up paying a price doing the sorting in the source system too, and you may also choose the wrong sort order (another reason for our use of ASDW). You can modify this time limit for compression with the "ProcessingTimeboxSecPerMRow" property of the service (a value of 0 forces to use all the data to find the best option, taking more time).
6. Hint for VALUE encoding on numeric columns.
   a. You can use this hint for all numeric fields (especially SKs): https://docs.microsoft.com/en-us/sql/analysis-services/what-s-new-in-sql-server-analysis-services-2017?view=sql-server-2017#encoding-hints
7. Hint for disabling hierarchies on SKs.
   a. See the section "IsAvailableInMdx" at https://blogs.msdn.microsoft.com/analysisservices/2018/06/08/new-memory-options-for-analysis-services/. This may significantly reduce the "columns hierarchies size" memory consumption. (For example, columns in fact tables don't need hierarchies, like having a "Total" column or ID fields as rows/columns (axes) on Excel's charts.)

   b. (This change is not reflected in this example, so additional savings could be achieved).

Third step: Optimizing fact tables

Similarly, we implemented the following Steps for optimizing Facts (the bigger tables):

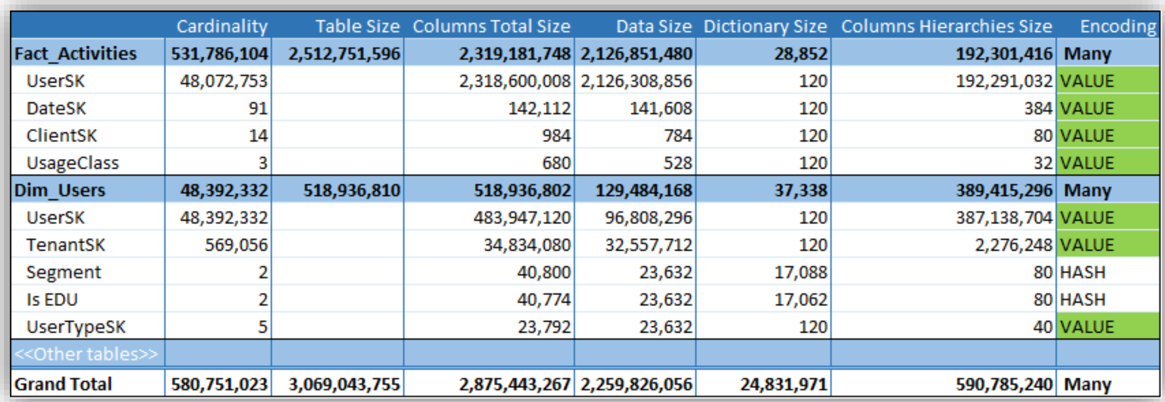
1. Handle early arriving facts.
   a. In a comprehensive data warehouse ETL process we would usually establish processes to handle fact records that reference dimension's members that didn't get into the dimension yet (because of real-time characteristics at source or data quality issues).
   b. For this example, we chose to discard the fact records without a corresponding dimension member. (You can see that Fact_Activities's UserID field has a cardinality of 48.7 million records, while Dim_Users' UserKey field has 48.3 million.)
2. Replace dimension IDs with their surrogate keys.
   a. After loading into ASDW, we joined the fact table with the dimensions that included the surrogate keys, and created a new version of it with only the integer surrogate keys. The view to feed the data into AAS used this table. (Having the original data along with the SKs in ASDW also allows for data lineage audits.)

   ***Note 7**: You should never include primary keys of fact tables in your Tabular model (being unique values, they cannot be optimized for compression), which is critical in big models.*

3. Reduce cardinality (data type conversions).
   a. Idem as with step 2 for dimensions explained earlier. This step is much more critical for fact tables (the bigger tables – See References for more).
4. Consider moving calculations to the source (to use in compression evaluations).
   a. If you can use a Measure with DAX instead of a column, definitively prioritize it. This way, the calculated value will only be done at query time over only the needed data, and not consume any storage.
   b. However, if you still have calculated columns that you must include in your model, then you should consider including it as part of the data ingestion instead of as a calculated column, so this data can be optimized for compression, as explained earlier.
5. Ordered by less diverse SKs first (to maximize compression).
   a. As you can see in our example, the fact table has ½ billion rows, but only 48 million unique user IDs, meaning that there will be repetition of values that could be optimized for compression. As we covered for dimensions, some testing is needed to help the engine compress as much as possible by using your knowledge of the data.
6. Increased Tabular sample size for deciding Encoding, by considering segments and partitions.
   a. For very advanced optimizations that we will cover in a future post, you can test alternative sizes for segments and partitions, as mentioned earlier (using the server property "DefaultSegmentRowCount").
7. Hint for VALUE encoding on numeric columns.
   a. Similar as with dimensions explained earlier.
8. Hint for disabling hierarchies.
   a. Similar as with dimensions explained earlier, but we can apply to other fact columns too and not just SKs.

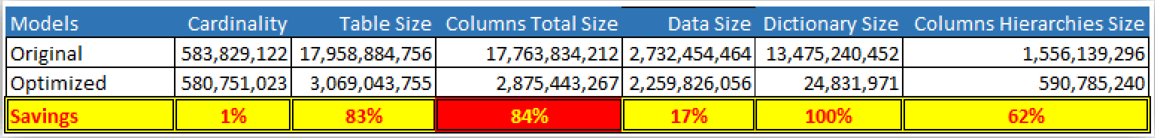
Results

After all these optimizations, we arrived to an AAS model like:



	Cardinality	Table Size	Columns Total Size	Data Size	Dictionary Size	Columns Hierarchies Size	Encoding
Fact_Activities	**531,786,104**	**2,512,751,596**	**2,319,181,748**	**2,126,851,480**	**28,852**	**192,301,416**	**Many**
UserSK	48,072,753		2,318,600,008	2,126,308,856	120	192,291,032	VALUE
DateSK	91		142,112	141,608	120	384	VALUE
ClientSK	14		984	784	120	80	VALUE
UsageClass	3		680	528	120	32	VALUE
Dim_Users	**48,392,332**	**518,936,810**	**518,936,802**	**129,484,168**	**37,338**	**389,415,296**	**Many**
UserSK	48,392,332		483,947,120	96,808,296	120	387,138,704	VALUE
TenantSK	569,056		34,834,080	32,557,712	120	2,276,248	VALUE
Segment	2		40,800	23,632	17,088	80	HASH
Is EDU	2		40,774	23,632	17,062	80	HASH
UserTypeSK	5		23,792	23,632	120	40	VALUE
<<Other tables>>							
Grand Total	**580,751,023**	**3,069,043,755**	**2,875,443,267**	**2,259,826,056**	**24,831,971**	**590,785,240**	**Many**

Which **resulted in up to ~84% improvement in compression!** (Given some limitations in our example, this could be further optimized thru "IsAvalilableInMdx" configuration to further reduce the "columns hierarchies size", as mentioned earlier.)



Models	Cardinality	Table Size	Columns Total Size	Data Size	Dictionary Size	Columns Hierarchies Size
Original	583,829,122	17,958,884,756	17,763,834,212	2,732,454,464	13,475,240,452	1,556,139,296
Optimized	580,751,023	3,069,043,755	2,875,443,267	2,259,826,056	24,831,971	590,785,240
Savings	1%	83%	84%	17%	100%	62%

This all means that, for this particular example, a combination of best practices in data modeling and tweaks to properties, we could achieve an ~84% compression to either fit 5 times more data into the same capacity, or reduce our resource consumption.

> ***Note 8**: in a comprehensive DW project, we should also handle many other data processing, quality, and integrity scenarios not covered here (i.e. late arriving facts, proper handling of early arriving facts, and more - See Kimball's link below for more details). In this example we only focused on compression techniques to maximize AS scalability.*

References

- Analysis Services documentation: https://docs.microsoft.com/en-us/azure/analysis-services/
- Analysis Services Team blog: https://blogs.msdn.microsoft.com/analysisservices/
- Kimball's DW/BI design techniques (dimensional modeling, star schemas, surrogate keys, ETL for Data Warehouses, etc.): https://decisionworks.com/data-warehouse-business-intelligence-resources/kimball-techniques/
- Azure SQL Data Warehouse IDENTITY: https://docs.microsoft.com/en-us/azure/sql-data-warehouse/sql-data-warehouse-tables-identity
- Using Polybase in Azure SQL DW: https://azure.microsoft.com/en-us/resources/videos/loading-data-with-polybase-in-azure-sql-data-warehouse/
- AAS scale-out: https://docs.microsoft.com/en-us/azure/analysis-services/analysis-services-scale-out
- AAS high availability: https://docs.microsoft.com/en-us/azure/analysis-services/analysis-services-bcdr
- VertiPaq Analyzer: https://www.sqlbi.com/tools/vertipaq-analyzer/
- Optimizing High Cardinality Columns in VertiPaq: https://www.sqlbi.com/articles/optimizing-high-cardinality-columns-in-vertipaq/