Do we really need Hash Keys?

During the community event 'Data Vault Day' in Hamburg, Germany, near the end of 2016 an interesting discussion emerged on the topic of Hash Keys – specifically about the merits of stepping away from the Hash Key concept.

As a brief recap, the Hash Key concept was introduced as part of the 'Data Vault 2.0' and means that the Data Warehouse key is an encrypted ('hashed') version of the Business Key.

A Business Key can be defined as a singular key, a concatenated key, a composite form or a complex combination of these, but regardless of its definition it can be hashed using a one-way encryption algorithm such as MD5 or SHA. The resulting single value is the Hash Key and can act as the Data Warehouse (Hub or Link) key.

Additional background information and context on Hash Keys is provided a bit further in this paper.

The discussion on *not* using Hash Keys was the result of a case study that raised issues concerning Hash Keys related to re-keying - where the source transactional system *recycles* its keys. For example, when a Customer Id from a long-gone customer is reused for a new Customer after a certain period of time.

If the Customer Id happens to be the Business Key, as it is in this example, then this is a problem. The source transactional system may not remember that there ever was a different customer mapped to the Id, but the Data Warehouse does not forget.

Managing this re-seeding of keys can be handled in different ways, but usually it is necessary to *uniquify* the keys.

The 'old' and 'new' customer - who share the same Customer Id - must be recognisable as separate instances of, in this case, the Business Concept 'Customer' (the Hub). To be separate instances they need to have unique Data Warehouse keys, even though the Business Key (the Customer Id) for both instances was provided as the same value by the source system at different points in time.

A simple way to achieve uniqueness is by defining the Business Key as a concatenated key, for example by combining the Customer Id with something that guarantees uniqueness.

Re-keying is not an issue that specifically applies to Hash Keys, but the topic of conversation was that Hash Keys complicate this issue because the original values are not immediately visible.

For example, the following (SHA256) hash outputs of the (concatenated) Business Key are not very informative:

- FBBEC44A6D388D8BBA9A212806DD111945AECDDB5100C46A1438B1467E5F51CF
- 607B96509C1949C20D440BB1FAE13B85B3A5694B5289F5A5CD6AE3D9FAECBBA2

If you would see the 'original' Business Key definition and values it is easier to understand what's going on:

- 35 + "< 2015"
- 35 + ">= 2015"

This is an example of Customer Id ‘35’ being defined as a different customer depending on a system date, a fabricated example of when the source system started recycling its keys.

To understand the original values, you need to query the ‘real’ Business Key which is available in the Hub. Otherwise, you can’t easily see what you did to ‘make things unique’.

It got me thinking about the various pros and cons of Hash Keys a bit more. After all, a Hash Key is nothing more than a representation of the original Business Key value.

Hash Keys in Data Vault

The introduction of hash keys as part of Data Vault 2.0 (DV 2.0) has been instrumental in achieving a mindset shift for Data Warehouse design, and credit goes to Dan Linstedt for making this an established practice.

Data Warehouse architectures have traditionally relied on the generation of sequence (integer) values to manage the Data Warehouse key distribution - the sequence key. This is sometimes referred to as ‘meaningless’ key – but I like sequence better because the key arguably does have meaning.

In the olden days of Data Warehousing, you had to make sure you updated your ‘key tables’ first *before* loading any tables that make reference to these tables (e.g. with Foreign Keys). For instance, a Dimension table had to be loaded before the Fact table.

This loading dependency also means that when you have to *reload* your key table (i.e. for refactoring purposes – driven by ongoing clarification of requirements) you also need to reload *all* the tables that depend on this table via the Foreign Key.

This is the concept of the Data Warehouse key distribution, which traditionally happened as part of a larger ETL process that also incorporated other concepts such as transformations and managing changes over time.

Data Vault, with its separation of concerns, has isolated the key distribution concept and embedded it into the Hub entity and associated loading pattern. This separated the key distribution concept and implementation from other mechanics such as handling changes over time.

In ‘Data Vault 1.0’ the key distribution mechanism was still implemented as a seeded (sequence) integer value, but in DV2.0 this concept and its implementation has been upgraded to using a hashed value of the Business Key.

Because of their deterministic nature, Hash Keys made it possible to load Facts before Dimensions or Satellites and Links before Hubs, thus removing loading dependencies. The Hash Keys also made it possible to truncate (and reload) a child table (i.e. Satellite, Dimension) without having to also reload the parent table (i.e. Hub, Fact).

Generally speaking, it was made possible to load data in essentially any order throughout the entire solution.

The application of Hash Keys has opened up our collective minds to implementing parallel loading - the ability to load data in any order and options to easily scale out across technical environments.

In Data Vault, the definition of a Hub does not *prescribe* using Hash Keys. This is a matter of separating concepts from physical design. The implementation of the key distribution concept using Hash Keys (or other approaches) should therefore be considered as options in the technical architecture.

The Hash Key explanation in this paper is limited to the Data Warehouse keys, so it covers the Hub and Link keys. I don't include hash diffs in this paper. A hash diff is the hashed value across attributes used for record change comparison. They use the same hashing technique but for a different reason.

In my view hash diffs are very useful to be able to support a single value comparison as opposed to a multi-column comparison. While we're talking briefly about hash diffs; just note that the diff doesn't preserve the attribute *order*. Something to keep in mind.

Issues with Hash Keys

Hash Keys have their own issues. They appear 'random' (which they are not, of course) to the user, and need to be joined to the Hub tables (when looking at context / Satellites etc.) to retrieve the original meaningful values.

To the RDBMS this apparent 'randomness' is reflected as well in the sense that hash output does not have a sequential nature - which can wreak havoc on indexes.

For instance, if you consider SQL Server as RDBMS the default for a Primary Key is to have a Clustered Index, which means the order in which the data are stored on disk is forced by the key (definition). The table on disk *is* the Clustered Index.

So, if your Hash Key is (part of) the Primary Key with a Clustered Index the order of the data (on disk) will need to be continuously updated because of this 'random' nature. This causes significant issues related to page splits and index fragmentation, which can cripple your performance after even a single set of inserted records.

You can change the index to a Non-Clustered Index, but this will mean your table becomes a heap unless you have a Clustered Index on another attribute. And the hash value is still the key you use to join tables together.

It seems to be a common experience that it is fairly easy to get information *into* a Data Vault, but it is sometimes problematic to get information *out* of it again. Indexing strategies may need to be geared towards the latter, and this is why some deployments have reverted back to using integer values. The consideration here is to 'take a performance hit' in terms of loading dependencies, but querying becomes easier again.

The implementation decision really comes back to the business requirement. Perhaps loading once or twice a day is enough to satisfy data availability needs.

It can be useful to consider that the approach and delivery thereof can (and arguably should) morph over time when requirements change, which is very likely to happen. Designing for this is the 'virtual data warehouse' mindset as advocated on my weblog www.roelantvos.com/blog. More on this later.

Hash Keys also tend to use up a lot of space, which is reflected in I/O costs when working with the data. In my experience, Link tables in DV2.0 are regularly one of the larger tables in the Data Warehouse (bigger than most Satellites), especially when they include more than two business concepts.

Each Hash Key depending on the algorithm used is typically 32 (MD5) or 40 (SHA1) characters - which are all used and don't compress well. Therefore, it is worthwhile to consider storing the Hash Key in a binary / raw format. The binary format will effectively half the required storage space (e.g. 16 bytes for MD5). GUIDs can be an option as well, but I personally felt the savings in storage did not outweigh the added complexities when joining to other platforms (e.g. the required conversion). If you use Hash Keys, binary storage is the way to go.

Hash collision is yet another relevant topic, but I won't discuss this further here. Have a look at Ronald Bouman's excellent article on the Hash Key and its algorithms. One of the core themes is whether a Data Warehouse team is the correct area to make decisions on risk (collision risk in this case), even though it may be very small. An interesting thought.

Lastly, there is a CPU processing overhead which applies to using Hash Keys - the algorithm has to be applied on every business key and combination of business keys.

Data Vault provides pointers to remediate this by suggesting values can be 'pre-hashed' in the Staging Layer, the source system (platform) or via distributed calculations. This limits the amount of time the algorithm has to be applied and / or allows processing to be distributed, but the calculation still has to be applied somewhere.

Evolution of these and other concepts is driven by continuous thinking, challenging and contributing to a growing knowledge base which deepens our understanding why we're doing things the way we do. We do this so we can make the right decisions depending for our specific scenario: *options and considerations*.

The consideration in this case is: would there be a way to maintain the functionality and mindset the Hash Key has introduced while making it work better? At least in some cases?

Do we need a Data Warehouse key at all?

First of all, it is interesting to revisit why we have a Data Warehouse key and the corresponding key distribution concept to begin with. Do we need a Data Warehouse key at all, as opposed to just using the natural key?

The short answer is still 'yes', but with the introduction of Hash Keys this may be in need of clarification again.

Sometimes it is forgotten why using sequence values for Data Warehouse keys have been around for such a long time, and discussions around not using Hash Keys tend to suggest using the source natural key directly (as in, no separate Data Warehouse key attribute). This would revert back to the early days of Data Warehousing, when solutions were sometimes implemented this way. But the key distribution concept was introduced for good reason.

Pairing a 'source key' to a Data Warehouse key is still considered best practice for various reasons including, but not limited to, having a consistent and fast way of joining (traditionally on integer values), avoiding variable length problems of keys, solving uniqueness and duplication problems, handling concatenation, composite and hard-coded values.

Having a Data Warehouse key separate from the identified key of the source system also allows you to adapt when required, for when there are issues with the source key (e.g. re-keying, duplication across systems and many more).

If we agree that we still need a Data Warehouse key separate from the Business Key (and I strongly believe so), there are two main established alternatives:

1. Using a sequenced surrogate key (usually an integer value)
2. Using a hash equivalent of the (combination of) identified keys – the Hash Key

The Natural Business Key, a new alternative

I would like to suggest a third option: a **'Natural Business Key'**. A Natural Business Key is the combination of the keys required to identify a given business concept stored as a concatenated value, including the required sanding.

Basically, it's a Hash Key without the actual hashing applied.

It is a single attribute created from the concatenation of the required Business Key attributes (including hard-coded values) and sanding delimiters. This also includes the handling of character sets, data type conversion and codepage (Unicode!) – the same as what would be needed as preparation for a Hash Key.

Let's investigate the benefits of using Hash Keys, and consider if we can maintain this functionality using Natural Business Key:

- Parallel / independent processing. Due to their deterministic nature, Hash Keys allow child tables to be loaded before parent tables. Sequence values don't support this, but this can be achieved using Natural Business Keys too.
- Cross-platform joining. Sequence values again don't support this as they are non-deterministic, but if we convert the Natural Business Key to an agreed fixed format (e.g. char, varchar, nvarchar) this works as well.
- Reloading / re-initialisation / refactoring / virtualisation. Same as above. Both Hash Keys and Natural Business Keys are deterministic, so they both produce the same results when rebuilding a table from a Persistent Staging Area.
- Pre-calculation / distributed calculation. The Natural Business Key would need less calculation than a Hash Key. Concatenation and NULL handling will be needed regardless, but the Natural Business Key is completed at that stage whereas the hash value requires the algorithm to be applied.

It seems worth looking into this a bit further.

Investigating Natural Business Keys in detail

My initial assumption was that, at the end of the day, Hash Keys would be faster on average when taking into account the occurrence of some very large (combinations of attributes for) business keys.

Some business keys would be small, and some would be very large. Since the hash output is always the same size, a Hash Key could still have a positive net performance upon comparison.

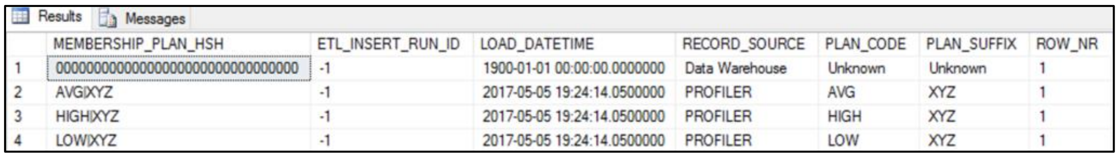
But I wanted to be more scientific about this, as it wouldn't be hard to calculate the average field length and use this to compare the sizes of the hash value and the Natural Business Keys.

The logic and metadata to calculate this is already available, so I did what every Virtual Data Warehouse follower would do: I tweaked the generation code (pattern) in the Virtual Data Warehouse application to create a 'Natural Business Key' version of the Data Warehouse.

The version of the Data Warehouse that uses the Natural Business Key as the Data Warehouse key can then be deployed as a separate version next to the version of the Data Warehouse that uses Hash Keys. The virtualisation approach allows the two instances (versions) of the Data Warehouse to be active simultaneously, which makes it easy to compare.

Of course, strictly speaking this is not really necessary since the length and storage requirements of Hash Keys is standard and can be easily derived. But I did it anyway, just because it's cool that it can be done and it took less than 30 minutes to implement this new feature, (regression) test it, validate if the Referential Integrity still holds up, generate the outcomes and deploy.

When you look at the output, in this case that of the 'Membership Plan' Hub, the result is as per the below:

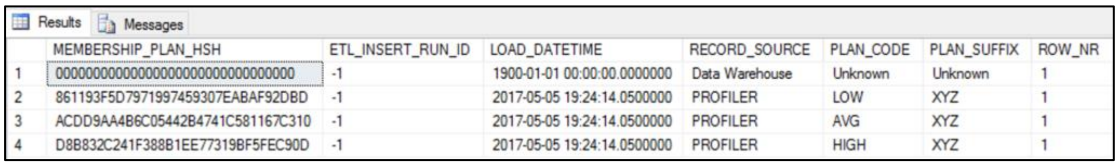


	MEMBERSHIP_PLAN_HSH	ETL_INSERT_RUN_ID	LOAD_DATETIME	RECORD_SOURCE	PLAN_CODE	PLAN_SUFFIX	ROW_NR
1	00000000000000000000000000000000	-1	1900-01-01 00:00:00.0000000	Data Warehouse	Unknown	Unknown	1
2	AVG|XYZ	-1	2017-05-05 19:24:14.0500000	PROFILER	AVG	XYZ	1
3	HIGH|XYZ	-1	2017-05-05 19:24:14.0500000	PROFILER	HIGH	XYZ	1
4	LOW|XYZ	-1	2017-05-05 19:24:14.0500000	PROFILER	LOW	XYZ	1

The above example shows the Natural Business Key as the single attribute of the 'Plan Code' and 'Plan Suffix' - an example of a composite Business Key. Note the inclusion of the '|' (pipe) sanding value.

The key with all the zeroes is the generic Hub zero record that is automatically generated.

For comparison's sake the original hash value version looks like this:



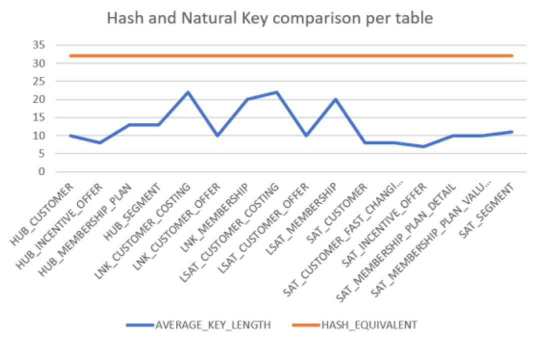
	MEMBERSHIP_PLAN_HSH	ETL_INSERT_RUN_ID	LOAD_DATETIME	RECORD_SOURCE	PLAN_CODE	PLAN_SUFFIX	ROW_NR
1	00000000000000000000000000000000	-1	1900-01-01 00:00:00.0000000	Data Warehouse	Unknown	Unknown	1
2	861193F5D7971997459307EABAF92DBD	-1	2017-05-05 19:24:14.0500000	PROFILER	LOW	XYZ	1
3	ACDD9AA4B6C05442B4741C581167C310	-1	2017-05-05 19:24:14.0500000	PROFILER	AVG	XYZ	1
4	D8B832C241F388B1EE77319BF5FEC90D	-1	2017-05-05 19:24:14.0500000	PROFILER	HIGH	XYZ	1

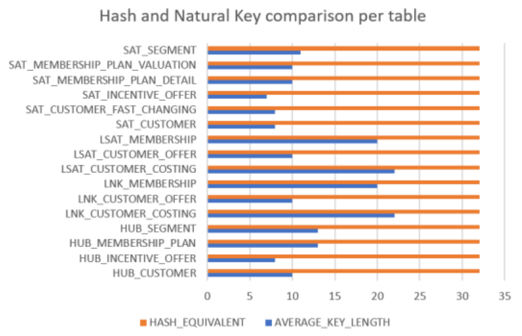
Comparison results

I created a validation script that calculates the average length of all the Data Warehouse keys in all tables within the Natural Business Key version, and compared this to the version that uses a Hash Key as Data Warehouse key. When running these scripts against the two Data Warehouse versions of my sample model and comparing the results the difference in required space is quite large.

The results are below.

TABLE_NAME	AVERAGE_KEY_LENGTH	HASH_EQUIVALENT
HUB_CUSTOMER	10	32
HUB_INCENTIVE_OFFER	8	32
HUB_MEMBERSHIP_PLAN	13	32
HUB_SEGMENT	13	32
LNK_CUSTOMER_COSTING	22	32
LNK_CUSTOMER_OFFER	10	32
LNK_MEMBERSHIP	20	32
LSAT_CUSTOMER_COSTING	22	32
LSAT_CUSTOMER_OFFER	10	32
LSAT_MEMBERSHIP	20	32
SAT_CUSTOMER	8	32
SAT_CUSTOMER_FAST_CHANGING	8	32
SAT_INCENTIVE_OFFER	7	32
SAT_MEMBERSHIP_PLAN_DETAIL	10	32
SAT_MEMBERSHIP_PLAN_VALUATION	10	32
SAT_SEGMENT	11	32





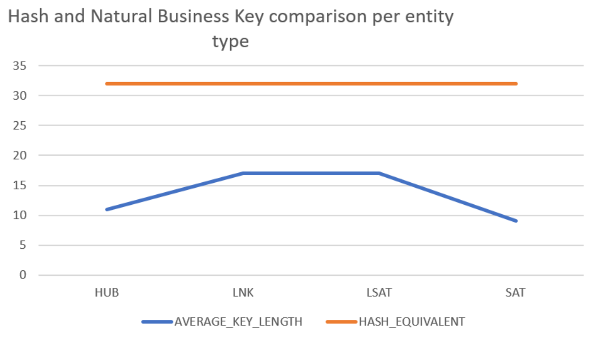
The output shows that within the TEAM / VEDW sample model there is no instance where, on average, the length of the Natural Business Key is higher than the hash value. The hash value in this case always 32 bytes (in this case character MD5 was used) so the average is, well, 32.

Using the recommend binary storage approach this value would be 16 bytes.

The sizes / lengths for the Natural Business Key are significantly lower, which will translate into reduction of storage requirements and better I/O without loss of functionality.

The following graphs show the relative size per entity type. As expected, the Hub and corresponding Satellites are smaller as they only contain a single Business Key. Links and Link Satellites are a bit larger, since by definition more than one business key is present.

TABLE_TYPE	AVERAGE_KEY_LENGTH	HASH_EQUIVALENT
HUB	11	32
LNK	17	32
LSAT	17	32
SAT	9	32



This is only a small sample of course, but looking at the systems I've developed there would not be many cases where the Natural Business Key size would be larger than the standard values of the Hash Key.

As a result, I believe that the Natural Business Key would have an overall net performance benefit in most cases. This really depends on the specific circumstances: if you have many really large values that need to be concatenated to form a Business Key the performance is expected to be worse compared to Hash Keys.

Having said that, I believe that in most cases the Natural Business Key may be a more efficient and clear way of developing up the Data Vault.

Options and considerations, as always

When using SQL Server, the Hash Key approach benefits from a fill factor with sufficient head room to keep index fragmentation in check (depending on your indexing approach).

A Natural Business Key would need this as well because its behaviour is still somewhat non-sequential, but usually better than the Hash Key (and obviously worse than the sequence value which is *really* sequential).

The Natural Business Key does allow for better planning on this as you can leverage the sequential nature of *some* of the business keys (scenario permitting). You can use this to your advantage if you add some of the more volatile attributes later in the attribute order - assuming you have to deal with concatenating multiple attributes to define a Business Key.

I haven't specifically considered Massively Parallel Processing (MPP) solutions, where the Hash Key could make for a good distribution attribute across nodes etc. And this is OK because as *always* (and I'm probably sounding like a broken record here) this is about options and consideration – making the right decision for your specific scenario and technical environment.

If you have adopted the Persistent Staging Area (PSA) concept (and why wouldn't you?) you will always have the option to change your mind later.

Having said that, I'm optimistic the Natural Business Key option can be considered in MPP environments as well. In the end, we're looking into options and considerations and we as specialists need to take our platform's strengths and weaknesses into account to define the right solution. We now have more alternatives for the implementation of the key distribution concept, and we can apply these where it makes sense.

I've been using this a lot and can't really fault it.

Do we *need* to choose at all?

If you adopt the 'Virtual Data Warehouse' mindset and its guiding principles for development, including a PSA, you have options. The Virtual Data Warehouse approach considers data solutions as assets that morph with the organisation over time and adapt to changing requirements and technology.

This 'morphing' is supported and implemented by (automatically) re-generating data loading (ETL) processes against a new or updated target model or definition, and automatically reloading the data against these new structures.

In other words, changing from a Hash Key to a Natural Business Key and vice-versa is basically just 'clicking a button'. The metadata of the data solution does not change, only the technical delivery – the pattern.

Based on the above it is perfectly feasible to optimise performance by *mixing* Hash Keys and Natural Business Keys in a single solution, and changing this mix automatically over time. With mixing, I mean that Hash Keys can be used for the few very large Business Keys (the outliers), while the smaller Business Keys can adopt the Natural Business Key approach.

The paper on 'Engine Thinking' available on www.roelantvos.com/blog/publications/ takes this one step further. This paper explains the concepts around automatic refactoring using key metrics related to data solution design and performance (aligned to requirements). Using these environmental variables, the data solution can automatically refactor to be the best fit for the specific circumstances.

If you think about it, the field lengths and RDBMS capabilities to deal with string values in joins can act as environment variables that can be measured and drive automated refactoring - to achieve the optimal performance mix for the physical model.

And then there are the exception cases where encryption itself is a requirement. Hashing is the default choice for this, it's why the algorithms exist in the first place. There may be a requirement for certain core data sets (Business Keys in this case) to be stored encrypted in a central location and the context elsewhere. This is metadata, and this can be automated.

There seems to be a feeling that while some the few very big Data Warehouse solutions (the multi-petabyte ones) benefit from adopting Hash Keys, many smaller deployments benefit from a simpler approach such as the Natural Business Key. Opinions strongly vary on this, and some of the largest systems have even reverted to using sequence values as Data Warehouse keys.

Instead of being opinions, these decisions can be truly data driven and the knowledge, measurements and patters can be fed into 'the engine' to improve the delivery of data solutions.

Keep sharing experiences with the community!