Data Vault 2.0 on Snowflake...

...to hash or not to hash... that is the question...

Data Vault 2.0 on Snowflake.

To hash or not to hash… that is the question

WARNING: Advanced Topic!

As data management, its principles and technology has evolved smarter ***data driven*** techniques have been developed to better use those technologies. One such data-driven innovation is the use of the hash-function.

Note: this article includes some introductory concepts to those not familiar with hashing, Snowflake architecture, Data Vault 2.0, PITs and Bridges which are labelled below. To skip these introductory discussions and get to the discussion in the topic above skip to "The Catch" below.

*"A hash function is any function that can be used to map data of **arbitrary size to fixed-size values**. The values returned by a hash function are called **hash values**, hash codes, digests, or simply **hashes**" - Wikipedia*

The outcome of a hash function appears like a random sequence of alpha and numeric, a single byte change to the source of the hash function can generate a completely different hash digest, thus hashing is a very effective method for distribution of data on a ***massive parallel platform*** (MPP) whose distribution of data is based on a key, see: bit.ly/3blXQXx. This also means that data with the same key will be in close proximity to each other based on that same hash/distribution key, they will have the same ***locality***.

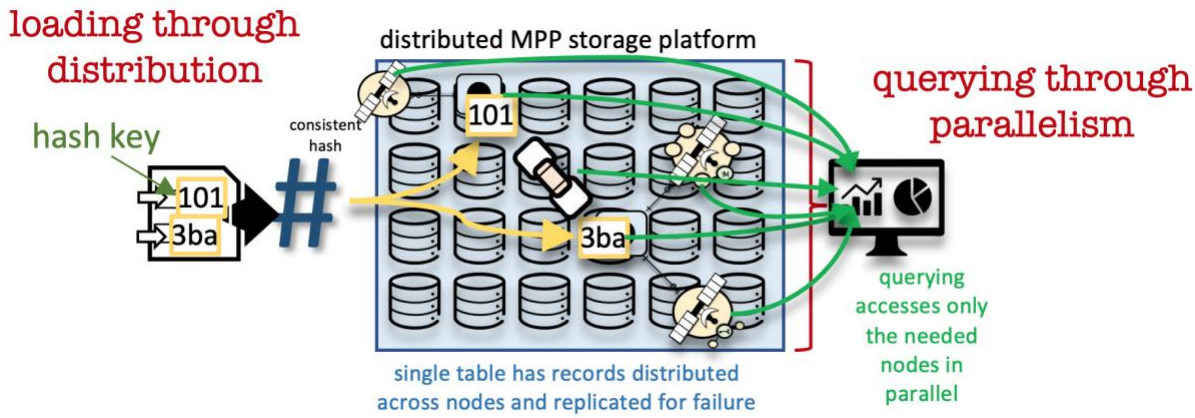


Figure 0-1 MPP distribution and parallelism

The same concept of distribution and locality with hashing is also used in the distribution of messages in Kafka that "randomizes" message packets across ***topic partitions***. Hashing with regards to streaming is based on speed, thus the algorithm used (murmur) is quick but does not guarantee there will be no ***hash collisions***, it doesn't need to. bit.ly/3bnTi2T

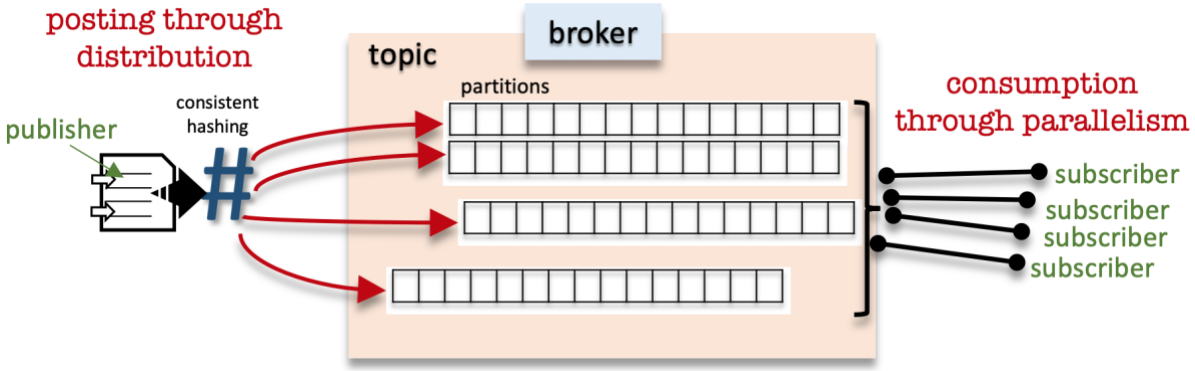


Figure 0-2 Message distribution and parallelism

Note that the same message hashed will always produce the same hash digest. It is consistent and the outcome is of data type ***binary***, always. Just remember, hashing is not encryption, although the outcome looks the same the intention of encryption is to decrypt the message digest at some point. See: bit.ly/3efn2kf

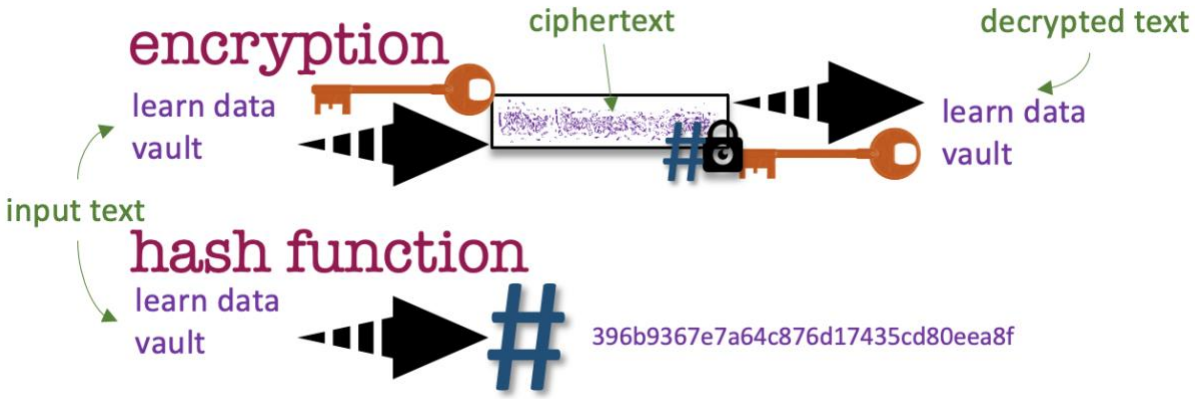


Figure 0-3 Hashing is not Encryption

Hashing in Data Vault 2.0

It is so that Data Vault 2.0 has also adapted to technology,

- First by dropping the need to have an ***end-date*** on the ***satellite*** table. This has to do with the recognition that update operations on a record are ***expensive*** and on ***immutable*** store it is difficult to do. Instead the end-date is ***virtualised*** with the use ***SQL window functions*** to infer the current record end-date by fetching the next start date (applied or load date) for a parent key (hub or link key), see bit.ly/2MSEY99.
- Second by making use of ***surrogate hash keys*** instead of ***surrogate sequence keys*** in all of data vault artefacts. This has profound effect on the loading pattern of related hubs, links and satellites, it's distribution on an MPP platform and the speed to query the data because of the data locality and spread of the data achieved when using this as a distribution key. It means that all related data vault artefacts can be loaded with no dependencies between them and in parallel, to elaborate on this point visit bit.ly/2II5fVt. Querying data vault content will also take advantage of MPP disk content locality because of the distribution, thus the SQL optimizer will *pull/query* thinner slices of disk bytes to satisfy query results.

There are more data vault 2.0 adaptions but let's keep this discussion focussed on hashing. Data Vault 2.0 based on surrogate hash-key hashes a business key that produces a consistent digest value as a surrogate hash key and takes advantage of MPP distribution and locality as previously described. But it is ***not*** a hashing dependent on speed, it is a hashing that ***must be unique to the business key*** used to produce the hash digest (surrogate hash key), otherwise you will end up joining content to a business key that does not belong to that business key. The surrogate hash key therefore ***cannot afford to have hash collisions***, thus an algorithm like murmur3 cannot be used, MD5 has shown vulnerabilities (see: bit.ly/3enR6HG) and therefore a minimum strength algorithm of SHA1 must be used. To better understand hashing in Data Vault 2.0 see bit.ly/371PykS.

As the volume of data increases so too does the ***depth*** of adjacent satellite tables to a hub or link table and the number of these satellites can grow too (raw and business vault). If performance with regards to querying these structures starts to become a bottleneck the way to deal with this is with the adoption of query assistance (QA) tables, and in data vault there are two such tables

- ***Point-in-time (PIT)*** table that in essence is a periodic snapshot of the relevant satellite keys and load dates around a hub or a link table that provides the optimized ***EQUI-join*** path for dimensional marts based on logarithmic PIT structures to the relevant satellite records (more on this click here: bit.ly/3ccwMcq)

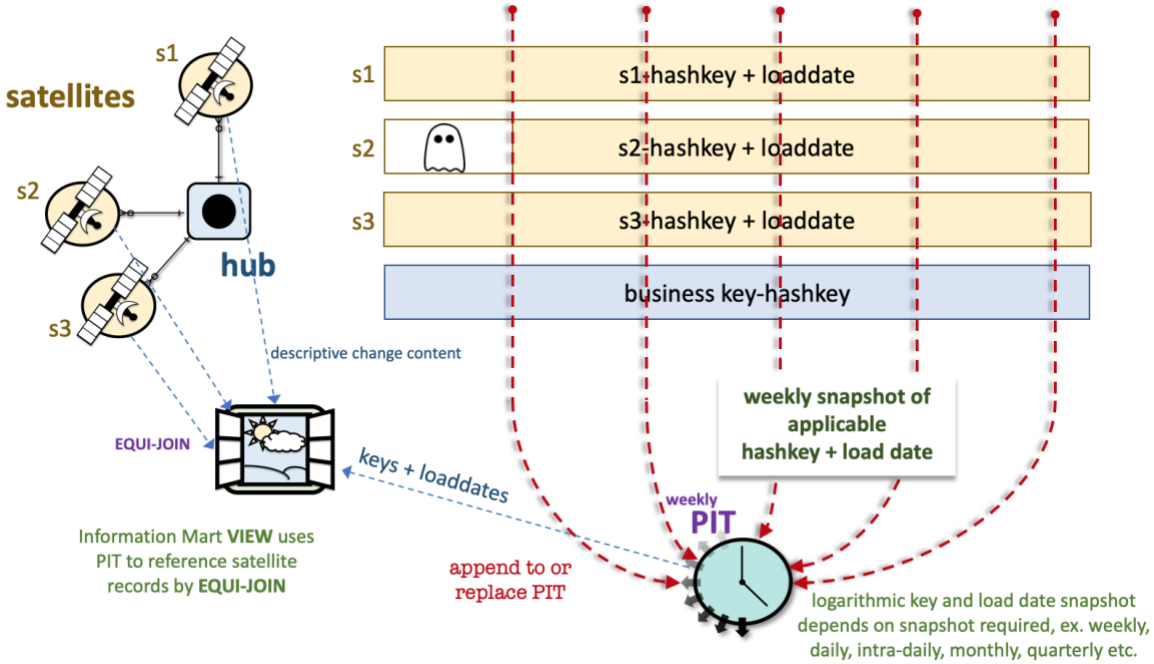


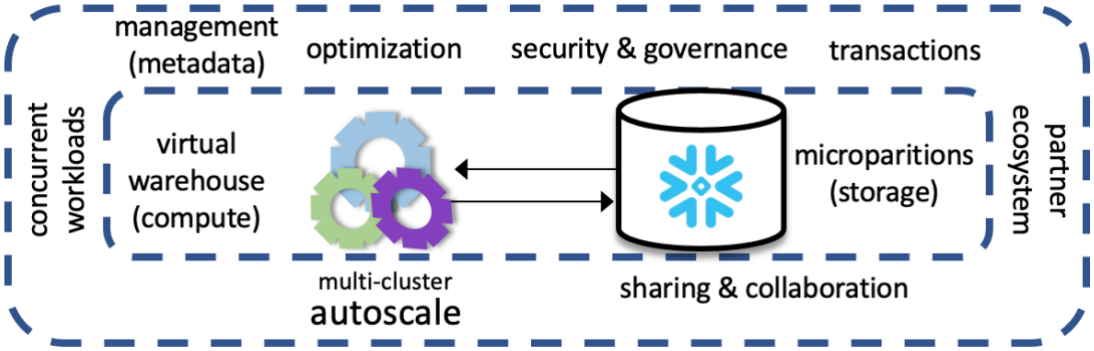
Figure 0-1 PITs are always tables, enabling EQUI-JOINs for Information Mart performance optimization

- ***Bridge table*** that shortens the join path journey from one hub to another hub based on a periodic snapshot frequency as well

Both these QA tables are disposable but only needed when ***performance suffers*** when querying the data vault. This performance pain is ***not*** disposed of, instead it is palmed off into the ***construction of PITs and Bridges*** rather than having the information mart view performance suffer. There is a technique to deal with the Pied Piper in this regard and it is described here: bit.ly/3iBfAzh

Now that we have an understanding of hashing and surrogate hash keys as they are used on MPP platforms to improve data vault 2.0 loading and querying performance, let's talk Snowflake.

Snowflake Architecture… in brief



Snowflake (www.snowflake.com) is described as an MPP platform (see: [bit.ly/3c13kpu](bit.ly/3c13kpu)), it achieves its massive parallelism by ***separating compute and storage*** through running virtual warehouses (as many as you like, see: bit.ly/3rkvF0y) and storing their customer's data on its proprietary ***hybrid storage of micro-partitions*** (see: [bit.ly/3rkv0fA](bit.ly/3rkv0fA)) respectively. This separation makes it ***massively parallel***, while uploading table micro partitions you can have

concurrent queries against that same data. Pre-sorted data is loaded and clusters naturally to that sort order, however if your querying of these tables will be in a different order to how the data is loaded then ***table clustering*** can be defined (see: bit.ly/2O7Y4bS) or the use of ***materialised views*** (see: bit.ly/30ipAWq) is recommended. However both options have an additional charge as both will perform ***reclustering*** to their own structures respectively when certain pre-conditions are met.

Overarching compute and storage is Snowflake's metadata services layer (using FoundationDB) that manages and tracks statistics on every table and micro-partition column level as data is ingested into the table for every customer of Snowflake (in addition to other services this layer offers). Queries that customers run against their data is evaluated through Snowflake's metadata and it knows where to find those micro-partitions that make up that customer's table because the location of the micro-partition column falls into a MIN/MAX range for that micro-partition column. The statistics Snowflake tracks are:

- Tables-level
    - Number of rows
    - Table size
- Micro-partition column level
    - Number of rows
    - **MIN/MAX values** (zone maps to enable query pruning)
    - NULL count
    - Distinct values count

Additionally queries that solely return one of the statistics listed above do not use virtual warehouses to return the results and instead those same statistics are made available to the customer's query. For an excellent demonstration on this and Snowflake's built-in caching see John Ryan's article here: bit.ly/2Oyvi3R

As metadata is used to retrieve micro-partitions for a table Snowflake does not employ index structures like the classic B tree index (see: bit.ly/2MWhMqw) to rapidly locate bytes or rows applicable to a query, it relies on the already-gathered metadata statistics to act as a lightweight index. Remember this point as we will revisit why this is important to the hash distribution of data in Data Vault 2.0.

Snowflake's micro-partitions are ***immutable***, they are ***never*** updated in place although update operations ***are allowed***. This paradox can be explained by understanding what happens at the micro-partition level for every table in Snowflake. DML UPDATE operations on Snowflake are actually ***copy-forward operations***, and (to borrow an Apache Cassandra term) if an UPDATE operation is performed on a row that record is given a ***tombstone*** (see: bit.ly/3rsYLed) and a copy of the same record is made active with the updated row values. The tombstone record now exists only in the ***Time-Travel period*** (see: bit.ly/3v4k8VE) as defined for that table, whereas the active record is available to the table not needing Time-Travel. It is why Time-Travel incurs a storage charge and why high-churn tables need a different table definition to that of a base table in Snowflake. Data Vault 2.0 fits this paradox very well because ***no update operations*** are permitted in Data Vault 2.0, only inserts.

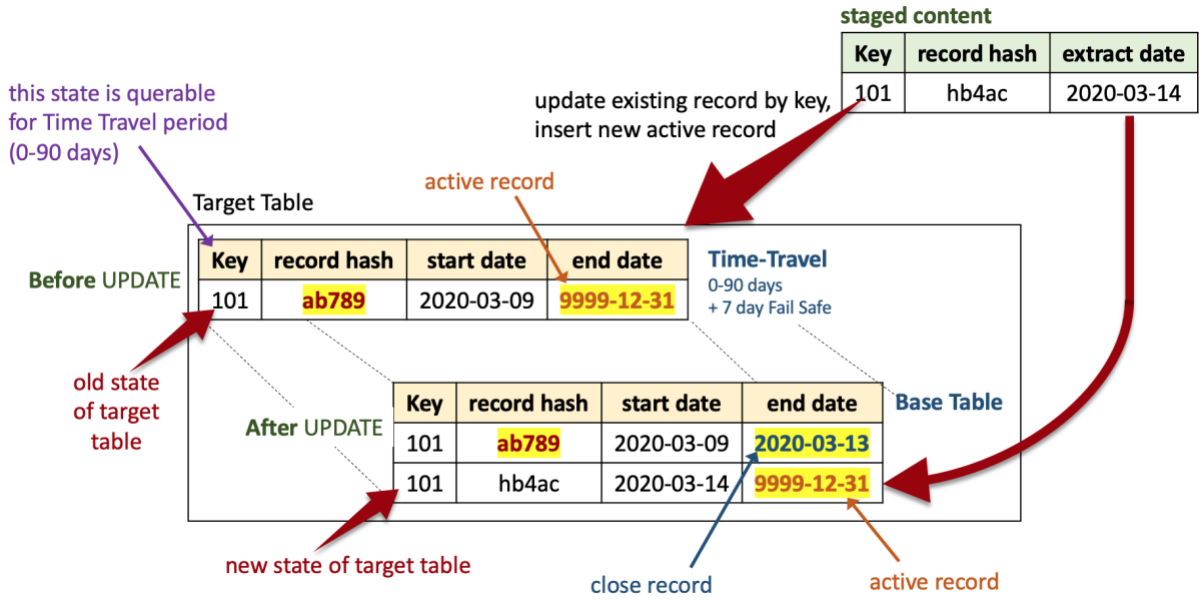


Figure 0-1 Time-Travel can be seen as a backup of the old state of a table for the TT period

Snowflake Elastic Data Warehouse White Paper, [bit.ly/3kQ2Wyp](bit.ly/3kQ2Wyp)

The catch

Kent Graziano goes into detail about surrogate-sequence key, natural key and surrogate hash key Data Vault implementations in his post here: bit.ly/3sUPc8f
The advice is this:

- A data vault based on surrogate ***sequence keys will perform the fastest*** when performing joins not only because of its use of integer data types to join but because it fits into the zone-map quite nicely. The downside to this style of data vault is the need to run table loads in a ***staggered*** fashion by forcing load dependencies between its tables. As described here and in Kent's blog you must load the hub table ***before*** loading its satellite tables; you must load all hub tables ***before*** loading the adjacent link table; and you must load the link table ***before*** loading its link-satellite.
  Fastest to query, slowest to load
- A data vault based on natural keys is the original data vault implementation and business keys are delivered as character text fields. This is the second fastest to query and all related artefacts can be loaded independently and at the same time. However it would start to become ***cumbersome*** to query when ***multiple columns make*** up a business key and if you add the additional data vault metadata tags like ***business key collision codes*** and ***multi-tenant ids***. The more columns in the join the more the join performance will suffer, something that having surrogate keys instead helps to alleviate.
  2nd fastest to query, fastest to load, but complex joins may suffer
- A data vault based on surrogate hashed keys ***must*** be stored as ***binary*** or raw data type. This will use half the byte length to that of varchar but still ***not be as fast as integer data types***. The byte length will always be the same in a surrogate hash key and it can be used in single column joins because the column values used as an input to generate the digest value can include any number of business keys columns as well as business key collision codes and multi-tenant ids and still produce the ***same***

byte length column. Its load matches that of a natural key based data vault but its joins might not be as fast as sequence key based data vault.
Pretty fast to join, joins are always consistent, fastest to load

Turn to dimensional modelling

Surrogate sequence keys are used in Kimball-style dimensional modelling for quite a different reason than in data vault, although the end goal is the same as to also achieve EQUI-join but through ***star-join*** queries. Kimball marts use surrogate sequence keys by assigning a new surrogate sequence key to ***every record*** loaded to a dimensional table and therefore the surrogate sequence key in dimensional modelling is ***time-based***, a fact record will tie to this single record in a slowly changing type 2 dimension (SCD Type 2), see bit.ly/3qjTILH and bit.ly/2MSj5GP.
This differs to a surrogate-sequence key based data vault in that a ***retained surrogate sequence key*** (durable key, bit.ly/3brn0lW) is always the same for the business key loaded, there is no time-based association to the surrogate sequence key in data vault… let's change that a little bit…
What I am suggesting here is different, and the clues have been discussed in everything up until this point.

1. Include an identity column in the satellite table (in addition to the surrogate hash key) – this column is auto populated as data is loaded for every record into the satellite table and has no relation to any of its adjacent data vault artefacts, meaning ***only satellite tables*** have their own independent identity/autoincrement column. Not its hub, not its link. Let's call the column ***DV_SID*** and prepopulate the ***ghost record with a negative key*** for that column.
2. Build PIT tables to query the data vault just like we build PIT tables on other MPP platforms that use indexes, except instead of including satellite surrogate hash keys and their load date timestamps we include the satellite table's autoincrement (sequence) column. The PIT table will now resemble a ***factless fact*** table as described under dimensional modelling, see: bit.ly/3c7TVfV and perform the most optimal of joins. The same logarithmic PIT windows apply!
   Fastest join, fastest load

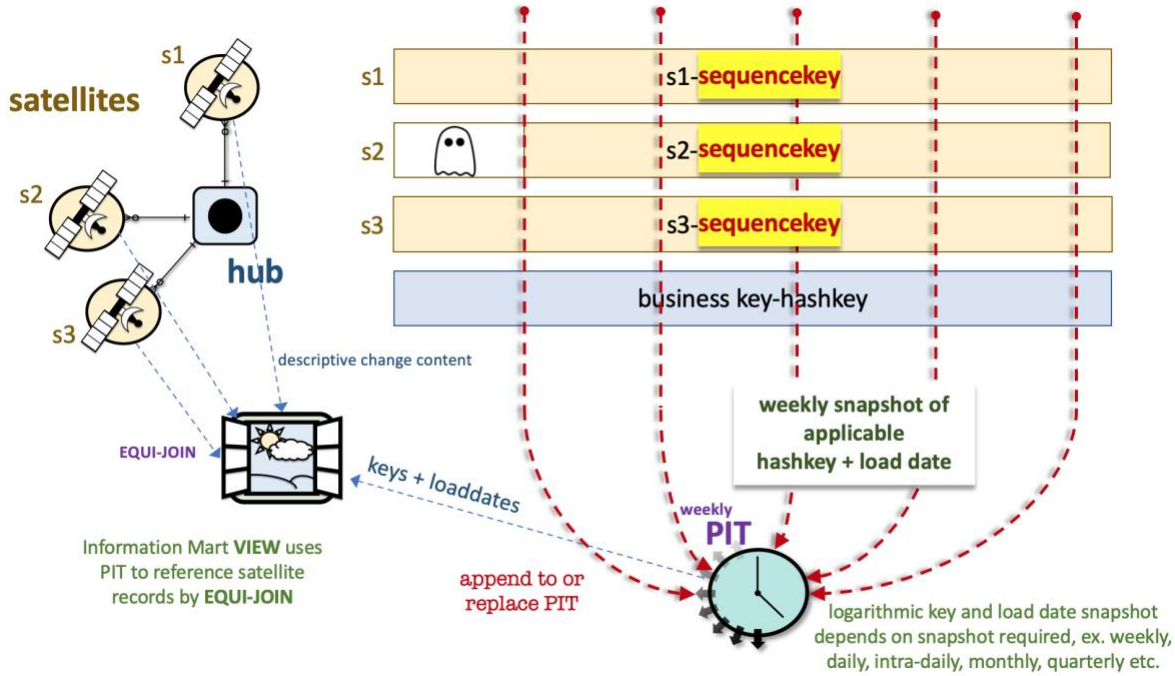
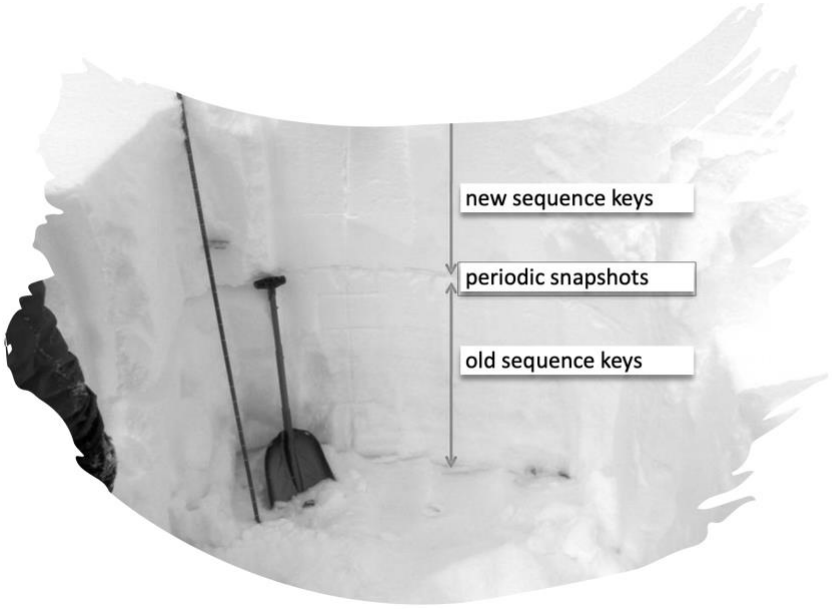


Figure 0-1 Factless PIT; or a SnowPIT

The PIT table in this instance is now a ***collection of sequence keys*** applicable to the ***snapshot period*** you need. Querying the satellites needed to build a PIT table will be faster because it is relying on the zone-mapping described where Snowflake's metadata will know the MIN/MAX range for the autoincrement column and know where to query the ***minimum needed*** micro-partitions to satisfy a query; which is also an integer data type. The same EQUI-join is achieved but with a ***temporal characteristic*** as in dimensional modelling; the PIT table now in this construction reduces the steps needed to build facts and dimensions based on Data Vault because it already has temporal sequence keys in the satellite tables.

Because PIT tables are logarithmic (monthly, weekly, daily), you could in fact add another additional ***helper*** column to the satellite tables to help with PIT joins. For instance, a DV_PIT_MONTHLY column that is of a data type integer with a value like 202105. Again, if the satellite is clustered by this column Snowflake's services layer will know where to find the micro-partitions to prune the query results.

To hash or not to hash… the answer is, yes keep hashing, adapt PIT and satellite tables to query data vault like a factless fact!



Snowpit

for Data Vault 2.0