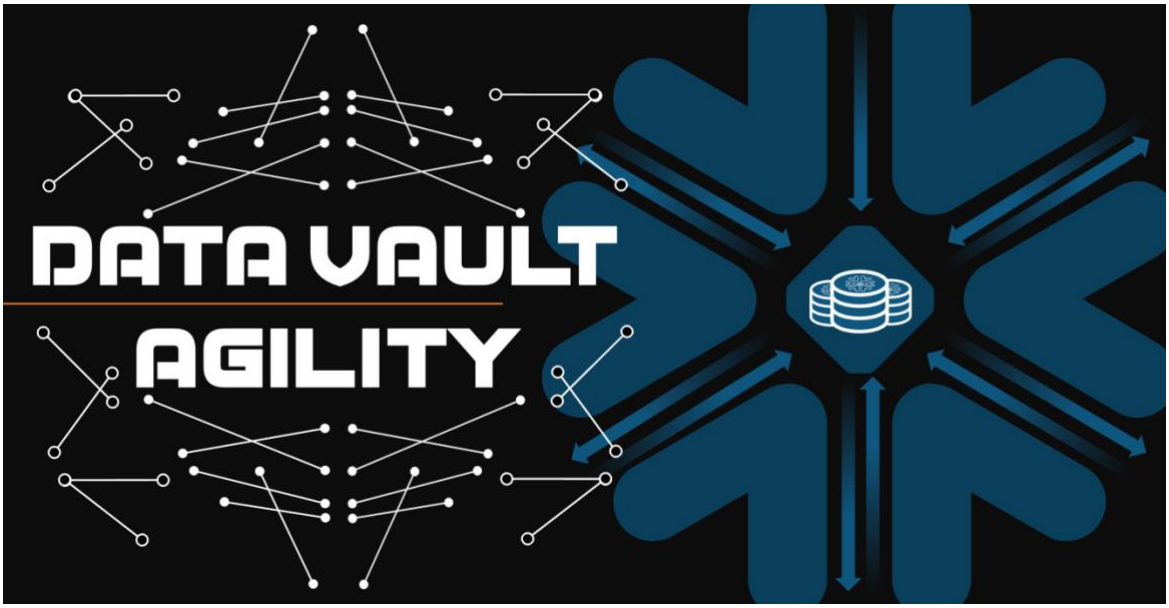
Data Vault Agility on Snowflake



Following on from my previous articles about Data Vault on Snowflake in the bullet points below. If you're already caught up, let's proceed directly with the purpose of this content!

- **Data Vault on Snowflake** (bit.ly/3dn83n8) ← *to hash or not to hash in Snowflake*
- **Data Vault Dashboard monitoring** (bit.ly/3CSP3aV) ← *using Snowsight to monitor the automated Data Vault test framework*
- **Data Vault PIT Flow Manifold** (bit.ly/3iEkBJC) ← *data driven PIT builds*
- **Why Equijoins Matter!** (bit.ly/3wohniH) ← *right-deep join trees and star optimized queries with PITs*
- **Data Vault's XTS pattern on Snowflake** (bit.ly/2Yt2yz6) ← *timeline correction and applied date*
- **Say NO to refactoring Data Models!** (bit.ly/3tPI66B) ← *the data model built to change*

Snowflake is *the* cloud data platform serving multiple types of workloads and a popular workload is loading data into a data vault on Snowflake. Yes, Data Vault implies repeatable patterns, but it is also up to *you* the architect to understand the **Snowflake architecture** and how we can leverage those for a Data Vault. In the blogs I have shown some techniques that help with optimizing your data vault, now let's get into how we can leverage some more technology this time to be able to load hub tables from all directions at the same time and still ensure hub table business key uniqueness. For an in-depth look into this concept visit:

- **A Rose by any other name… wait… is it still the same Rose?** (bit.ly/3xlFK0s) ← *business key collision codes (BKCC) to ensure passive integration from multiple sources*. The same key from a single source system but multiple files should have the same business keys, if we're lucky, the same business key representing the same business object exists across source systems. If a clash could occur, then we use BKCCs to ensure hub table uniqueness.

So, you see, the hub table is the ***integration point***, the focal point if you will be ingesting **multiple sources** at *any cadence* and recognize all link and satellites tables are simply

describing details about the hub, **the business object**, the thing the **business capability** is based on that make up a business.

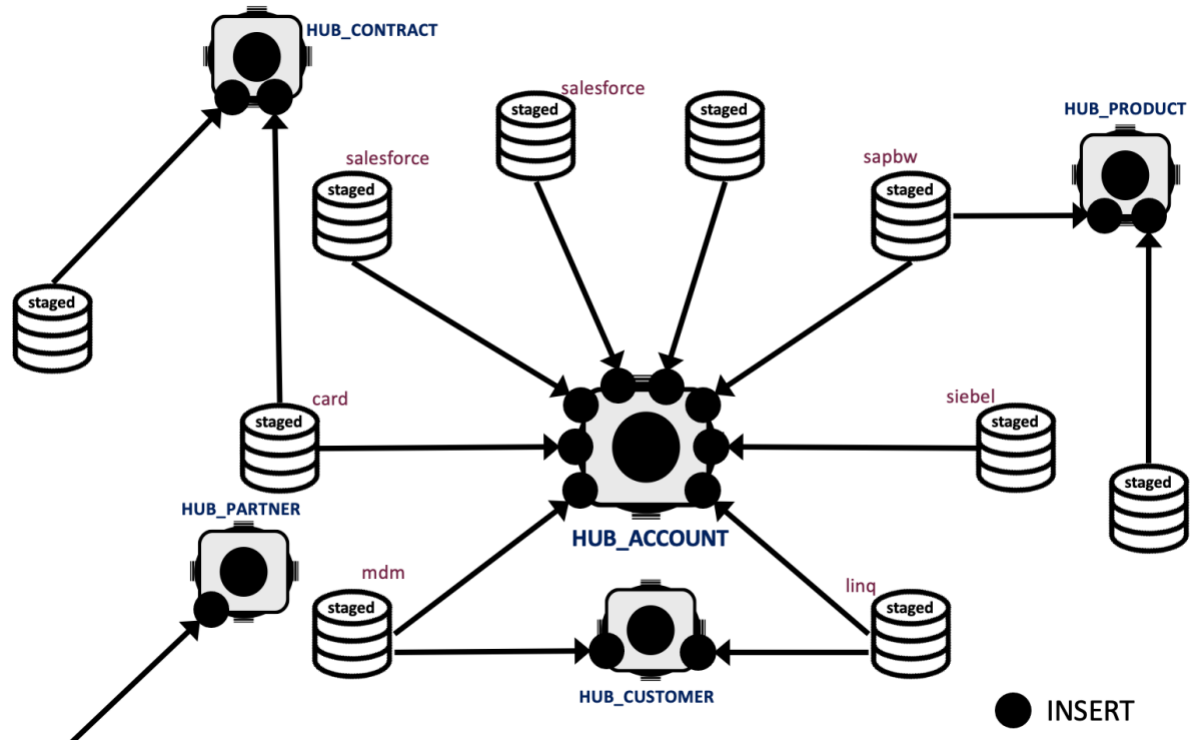


Figure 1 a micro view of integration; a Hub table is the effective centre, in Data Vault it represents the Business Object

What of Snowflake architecture do we need to know here?

Snowflake is implemented at **READ COMMITTED** transaction level, what this means is that users of data in Snowflake only see data as it has been committed by any other party on the platform. Snowflake does not lock entire resources for SQL INSERT operations, and this allows multiple parties to INSERT partitions to a single Snowflake table resource at the same time. The advantages to this approach are *massive*, it means that the Snowflake table object as a whole can be at a state of **eventual consistency**. Interestingly, transaction isolation level is offered as a parameter but with (currently) only one option… *perhaps some future change is on the cards?(see: bit.ly/3pyY3O7).*

- Snowflake Transactions, bit.ly/3Hlc3ks

	Dirty Read	Non-repeatable Read	Phantom Read	
READ UNCOMITTED	Possible	Possible	Possible	
READ COMMITTED	Not Possible	Possible	Possible	←SNOWFLAKE
REPEATABLE READ	Not Possible	Not Possible	Possible	
SERIALIZABLE	Not Possible	Not Possible	Not Possible	

Table 0-1 One of the ACID properties, Transaction Isolation Levels

In brief (learn more here: bit.ly/3Hp8SIq):

- **Dirty Read** – being able to read uncommitted transactions
- **Non-repeatable Read** – reading a row twice during a transaction may have differing values
- **Phantom Read** – rows are added or removed by another transaction during your transaction

For a data vault satellite table this is not of a concern because satellite tables are **single source**. We do not mix multiple source data into a single satellite table, especially raw vault satellite tables. Satellites are always single source, **INSERT-ONLY** tables. **The problem** however could occur at the hub and (less likely) the link tables, they are source system **integration points**, the only place in data vault where we conform source-system columns names, i.e., **the business key**. Because you can have multiple sources hitting the same hub table load there could be a race condition, an insert operation designed to leave the hub table as *a unique list of business keys* could be compromised if the *same* operation is executed from multiple sources at *the same time*. Especially true if multiple loaders are attempting to load overlapping business keys. Snowflake does support syntax for referential integrity but **does not enforce** them, i.e., the PRIMARY KEY and UNIQUE clauses on a Snowflake table object are **informational** but are useful for **reverse engineering** into modelling and BI tools. *See: bit.ly/3sNBDKX*

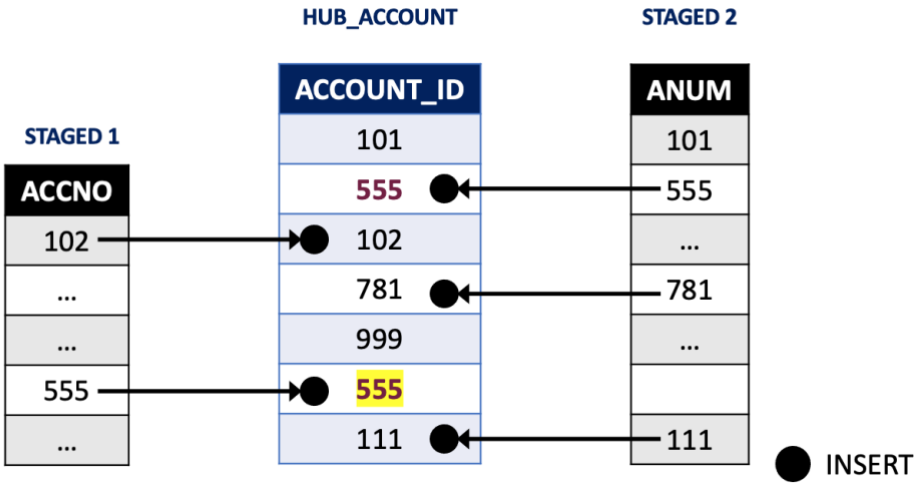


Figure 0-1 Snowflake's allows concurrent SQL INSERT operations to the same table; this will fail hub test for uniqueness

How do we then solve this for Data Vault?

"It is necessary sometimes to take one step backward to take two steps forward." - Vladimir Lenin

As you know **Data Vault 2.0** is an INSERT-ONLY architecture, and for good reason. Specifically in Big Data architecture performing updates is near *impossible* as we are dealing with immutable data stores. Reprocessing an entire blob file to make a change becomes very costly, *and why I don't advocate for refactoring ever!* Snowflake is the same (see: bit.ly/3mF37Pf), SQL UPDATE operations create new micro-partitions just as SQL INSERT operations do but since a Snowflake table is made up of hundreds to thousands of immutable files to the user an update appears as if it is updating data rows itself, *it is not*. The previous state of a record is committed to **Time-Travel** (see: bit.ly/3v4k8VE) and the updated record appears as the **active partitions** to the user. Therefore, it is fair to assume that SQL UPDATES are expensive and especially so on large and wide satellites tables, *hub and link*

tables on the other hand are neither large nor wide... keep that in mind, we'll come back to this later.
Snowflake's READ COMMITED isolation level means that one or more hub-loaders could be loading to the same hub table at the same time. Neither will lock the whole table and therefore the same business key (with BKCC and tenant id) could be loaded at the same time. To ensure only one process performs actions against the common table one at a time we need to somehow lock that table for INSERT operations while a randomly chosen secondary (or third and so on…) process can attempt to load to the target table when it is its turn. Luckily in Snowflake such a facility *does* exist.

1. Transactions

All SQL operations are executed with an AUTOCOMMIT in Snowflake and combined with the fact that table **metadata statistics** are **always up to date** it makes the cloud data platform extremely compelling as a scalable data solution. You do have the option of collectively grouping multiple SQL operations by explicitly setting a BEGIN and COMMIT boundary to committed SQL statements. This does give you the option to ROLLBACK if any of the SQL operations in that group of operations fail, however it does **not** lock resources for exclusive SQL operations.

resource	type	transaction	transaction_star	status	acquired_on	query_id
DATAWAREHOUSE.DATAVAULT.HUB_CUSTOMER	PARTITIONS	1640828018...	2021-12-29 ...	HOLDING	2021-12-29 ...	01a148bd-3...
DATAWAREHOUSE.UTILITIES.RECONCILE_HUB_ACCOUNT	STREAM	1640828016...	2021-12-29 ...	HOLDING	2021-12-29 ...	01a148bd-3...
DATAWAREHOUSE.UTILITIES.RECONCILE_HUB_CUSTOMER	STREAM	1640828018...	2021-12-29 ...	HOLDING	2021-12-29 ...	01a148bd-3...
DATAWAREHOUSE.DATAVAULT.HUB_ACCOUNT	PARTITIONS	1640828017...	2021-12-29 ...	HOLDING	2021-12-29 ...	01a148bd-3...
DATAWAREHOUSE.DATAVAULT.HUB_ACCOUNT	PARTITIONS	1640828020...	2021-12-29 ...	WAITING		01a148bd-3...
DATAWAREHOUSE.UTILITIES.RECONCILE_HUB_ACCOUNT	STREAM	1640828017...	2021-12-29 ...	HOLDING	2021-12-29 ...	01a148bd-3...
DATAWAREHOUSE.UTILITIES.RECONCILE_SAT_CARD_TRANSA...	STREAM	1640828022...	2021-12-29 ...	HOLDING	2021-12-29 ...	01a148bd-3...
DATAWAREHOUSE.DATAVAULT.HUB_ACCOUNT	PARTITIONS	1640828020...	2021-12-29 ...	HOLDING	2021-12-29 ...	01a148bd-3...
DATAWAREHOUSE.DATAVAULT.HUB_ACCOUNT	PARTITIONS	1640828023...	2021-12-29 ...	WAITING		01a148bd-3...
DATAWAREHOUSE.UTILITIES.RECONCILE_HUB_ACCOUNT	STREAM	1640828020...	2021-12-29 ...	HOLDING	2021-12-29 ...	01a148bd-3...

Figure 0-1 Showing locks, see: bit.ly/3pCKM70

2. MERGE WHEN MATCHED THEN UPDATE

Snowflake has the option to adjust the lock timeout parameter, that is, inherently without needing an external semaphore or resource pooling routine Snowflake will attempt the SQL operation again on a **locked shared resource** if it was not able to lock that resource initially for an update. You can adjust the parameter here, bit.ly/3mG0AEj, the default is 12 hours, but perhaps for a session this should be set to a much lower value considering we are only updating a hub or a link, a continual lock may be indicative of a deadlock state.
A SQL UPDATE or DELETE operation will lock the entire resource, *why?* Because Snowflake does not know which micro-partitions that make up the table resource these operations will be applicable to until execution time, a SQL UPDATE or DELETE will lock the target Snowflake table. A concurrent SQL UPDATE or DELETE operation will see the lock and wait for its turn to perform its SQL UPDATE or DELETE operation.
Snowflake's SQL MERGE statement has the potential to lock the whole table, see: bit.ly/3sFE4iE. The statement itself can be used to perform an SQL UPDATE, DELETE, or INSERT and *here's a little trick…* using the SQL MERGE statement does indeed **lock** the resource even if you are only performing an INSERT operation!
Now, *if* we allow for table updates to hub (and link) tables *only*, then what columns should we allow for updates in those tables? *The clue is in the standards itself! See: bit.ly/3exVCVD.*

- **10.3 Hub and Link Last Seen Dates (DEPRECATED)**

Yes, if we are going to enable multiple sources loading to a **shared resource** while still maintaining hub table business key *uniqueness* then why not reinstate the old "Last Seen Date" data vault metadata tag updates?
Because Snowflake tables are made up of micro-partitions (typically much smaller than parquet partitions) the cost to perform updates *is minimal*, and in terms of storage (and those persisted to Time-Travel) *are negligible*. Depending on the time-travel period you have set for the hub and link tables that additional storage would have accumulated from 0 to 90 days and thereafter *dropped off*. *In an INSERT-ONLY methodology just how much Time-Travel do you really need?*

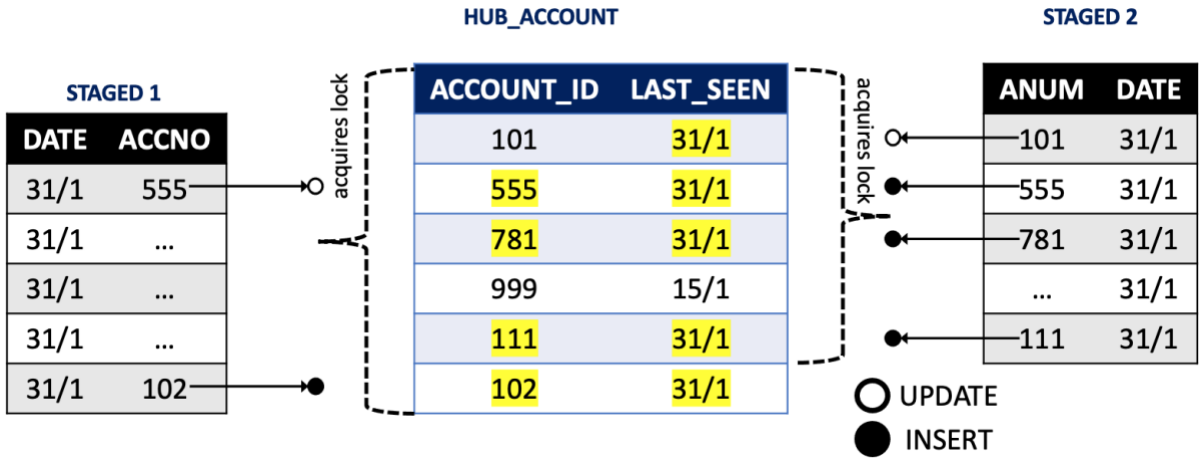


Figure 0-2 Acquiring locks in Snowflake essentially means each SQL operation must wait its turn to run its update

Data Vault 2.0 being completely INSERT ONLY did make available an alternative satellite table structure to track the *last seen date* of business keys or relationships. These are:

- **Record Tracking Satellite (RTS)** – tracks the occurrence of the business key (if the RTS parent table is a hub) or relationship (if the RTS parent table is a link)
- **Status Tracking Satellite (STS)** – tracks the appearance and disappearance of a business key or relationship when the source is a snapshot.

If we allow for updates to hubs and links alone then what do we do with RTS and STS? Are they still valuable to a Data Vault implementation?

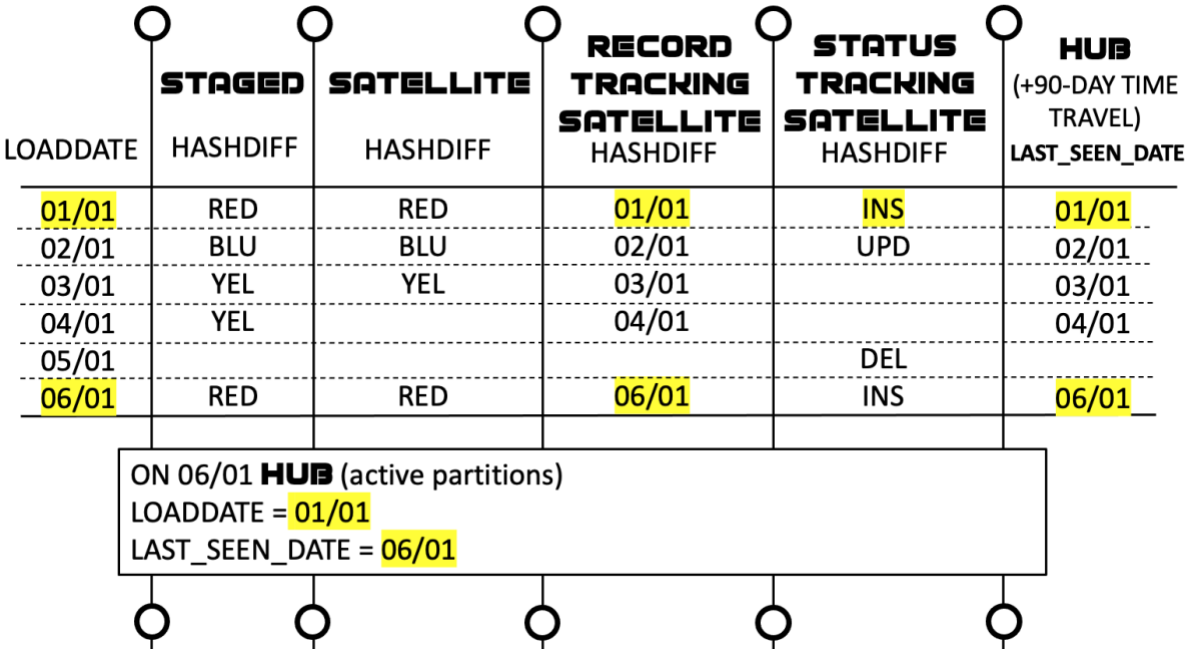


Figure 0-3 tracking per parent key, last seen date from another source may have an entry for 05/01

The **hub** table's LOADDATE will track the *first appearance* of the **business key** and the LAST_SEEN_DATE will track the *last time* the business key was seen, *nothing* in between. The **link** table's LOADDATE will track the *first appearance* of the **relationship** and the LAST_SEEN_DATE will track the *last time* the relationship was seen, *nothing* in between. RTS and STS are still INSERT ONLY but tracks more data about business keys and relationships while last seen date in hubs and links remains *limited*. From the illustration above RTS provides for **each instance** the parent key appears; STS will track the deletion of the parent key (only if the source is a snapshot) and hub tables only provide the first and last occurrence with no deletion detection in between. *The best outcome really is if the source sends a deletion flag! But these are the tools we have in Data Vault!*

Driving Key and Effectivity

If the data source **does not** contain a **business date** tracking the effectivity of the driving entity of the relationship, or you wish to track effectivity of a different driving entity than that of what is tracked in the data source then the need for an **Effectivity Satellite (EFS)** arises, see: bit.ly/3oS4k70. Yes, a LAST_SEEN_DATE column in the link table will give you what the **current** relationship is without needing one of the most complex Data Vault patterns, let's explore by way of an example.

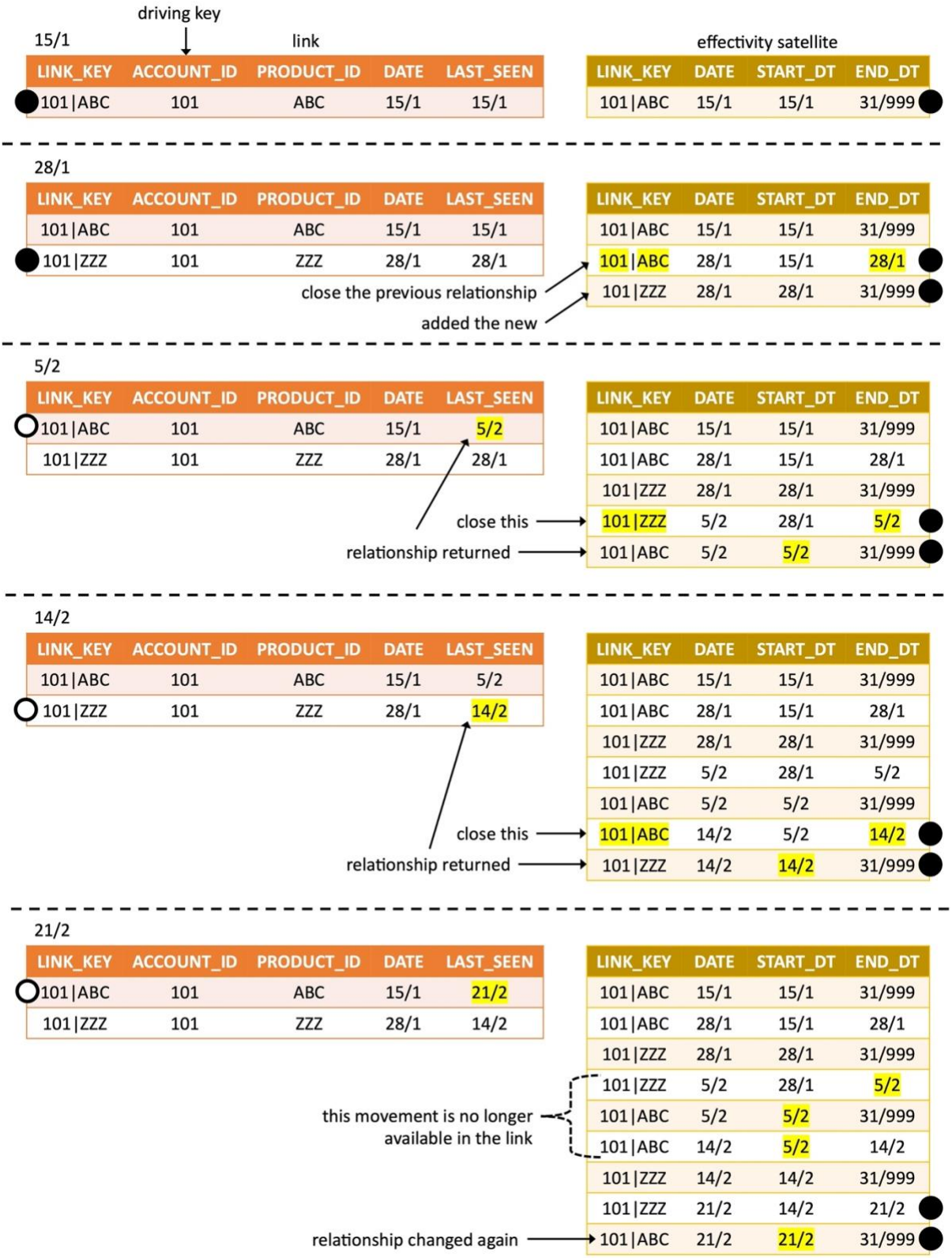


Figure 0-1 Driving Key & Effectivity v Link LAST_SEEN_DATE, link historical movement subject to Time Travel

No matter what the **driving key** is when utilizing the LAST_SEEN_DATE you will get the **current** active relationship for that driving key/relationship. It also does **not** require that you deploy **multiple** effectivity satellites for each driving key you want to track on a *single* link table. However, you will **not** be able to trace the **historical** movement of that driving to non-

driving key relationship, that is the exclusive realm of the Effectivity Satellite. *Perhaps for your use case, you do not need the historical movement…?*

What about the eXtended Record Tracking Satellite (XTS)?

bit.ly/3y4mUdV

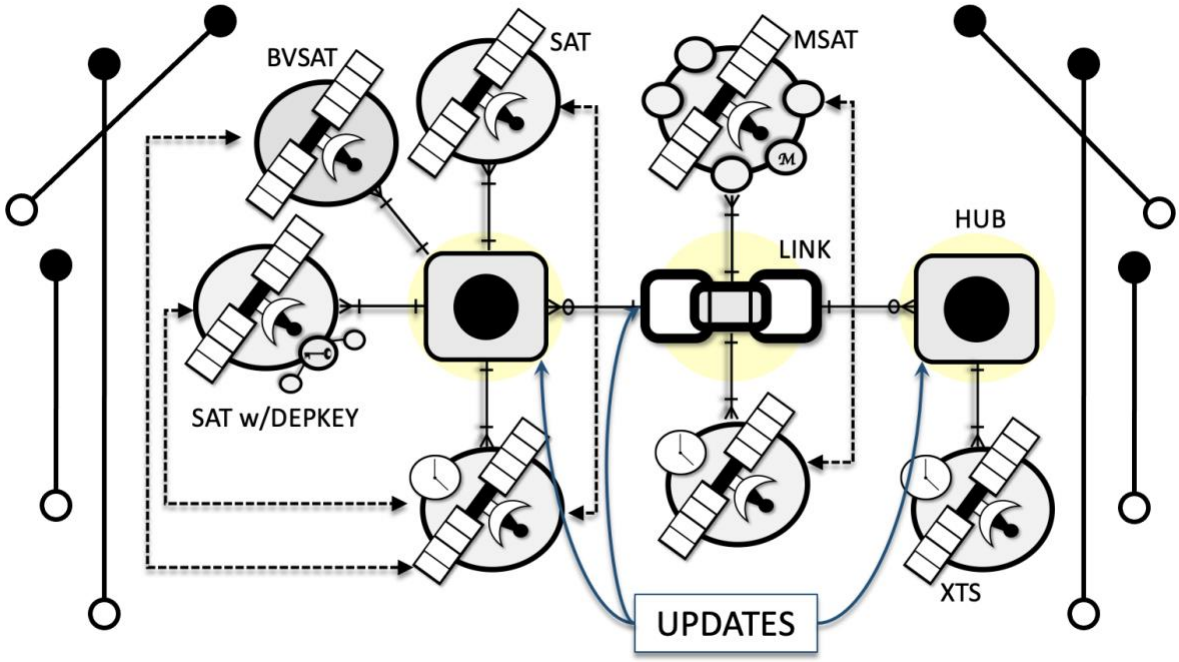


Figure 0-1 Data Vault with XTS

Like the hub table an XTS satellite will have multiple sources loading to it (and using it), however it differs in purpose *and grain* to that of the hub and link table. Because the satellite tables it supports are **single source** the insert to XTS *is isolated* by source as well, there will never be an overlap between source systems and files loading to a common table, the XTS table. By definition there is no need to lock the XTS satellite table and why Snowflake's transaction isolation level suites XTS so well.

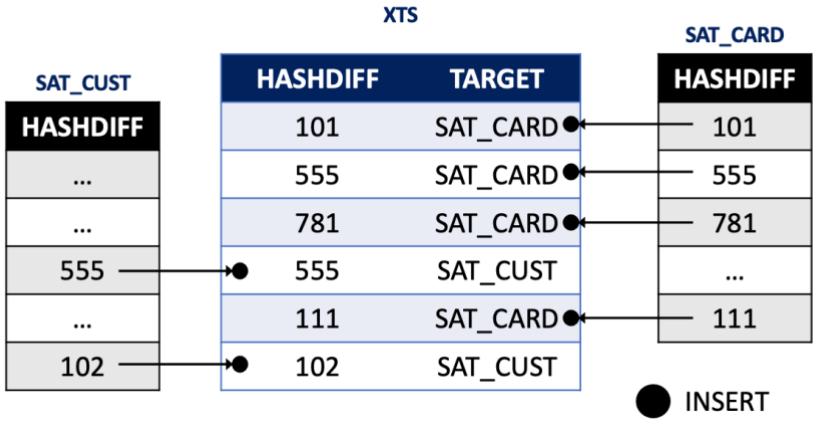


Figure 0-2 XTS entries never overlap! (Single parent key example)

The Bigger Picture

We have shown how the data vault model and the inherent Snowflake architecture can be leveraged natively to support ***passive integration*** and ***eventual consistency*** while still supporting **ACID** transactions; the alternative of course would be using resource pools and/or

external semaphores to restrict hub table updates to a single thread operation. The choice is ultimately yours; do you use an orchestration tool to manage resource locking, or do you allow the technology to natively manage that for you in a single tool?

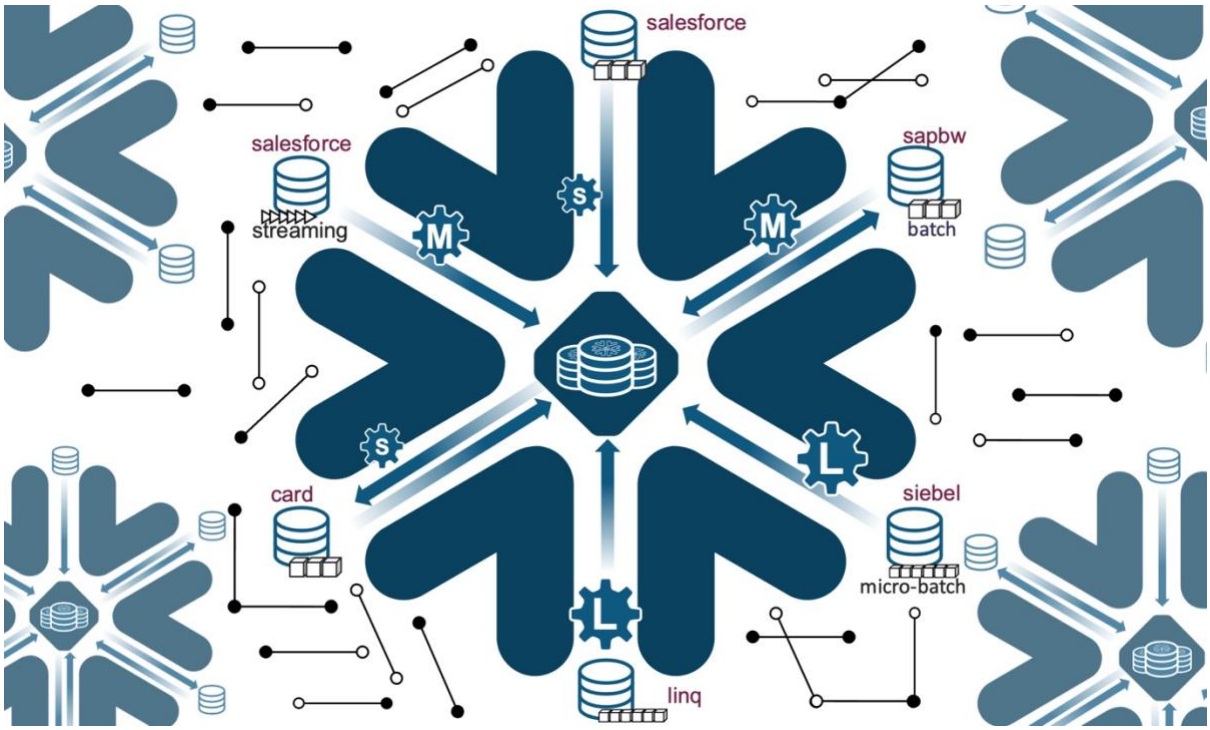


Figure 0-1 a macro view of integration, including Snowflake Sharing

The Data Vault Test Automation framework also contains integrated tables, recording loading statistics and errors from your Data Vault implementation but does not need table locking either! *See: bit.ly/3dUHPIS*

**Please note that Snowflake has soft limits to the number of concurrent locks in a wait state operation on a shared resource, reach out to Snowflake Support or your Account Executive to request to have these soft limits adjusted if needed.*

Multiple data pipelines are loading hubs, links, and satellites in isolation, *what if the speed of our processing does not meet the speed at which the data is being landed?*

Multiple source system cadence and how to handle it

Now this might be the final piece of the puzzle, handling data that arrives at any cadence. For this portion of the article, we will revisit Snowflake's Streams and Tasks. Recall that we used Snowflake's Streams on top of the Data Vault tables themselves to track **new data** that has been loaded and subsequently those metrics were made available to your **Snowsight** dashboards. Streams use an offset on a data object being a table, external table, share or even a view; we use them to add Data Vault Metadata Tags (record source, applied date, record hash etc.) before loading, and you should!

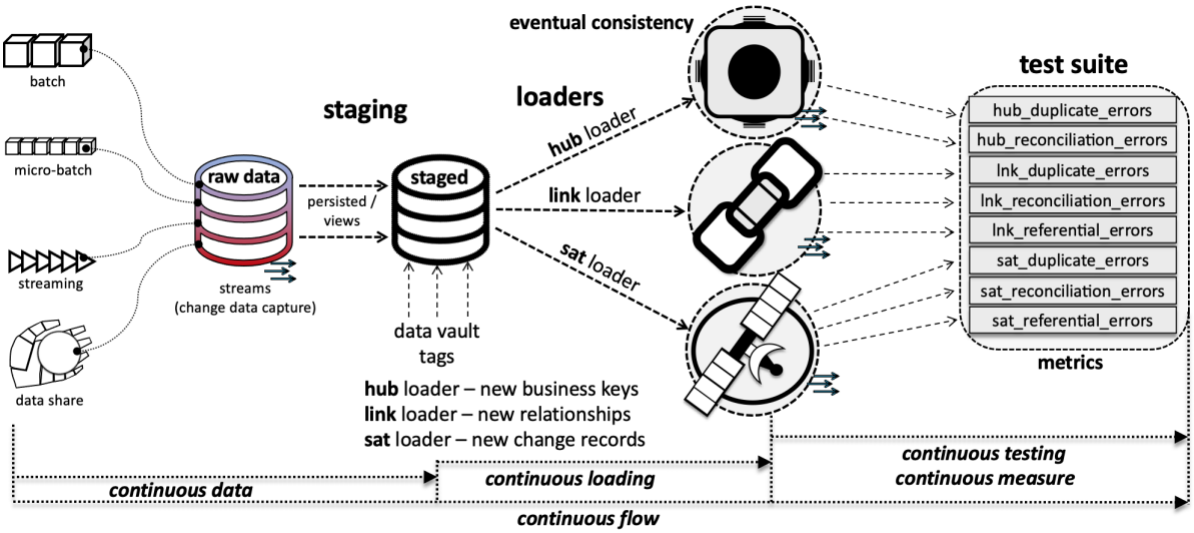


Figure 0-1 Streams keep the amount of data being processed to a minimum

The problem is that if you do not pick up the correct cadence than you could be trying to load more than one parent key update than your satellite load code will allow, and by allow, I mean it will miss data in the load. *Inconceivable!*
The solution may be far simpler than you think!

STEP 1: Set up a **Snowflake Stream** on the resource so that there is an offset set on that resource. *Recall* that you can have as many streams on a single resource as you like, and that the **offset** of a stream only moves if it is used in a **DML** statement. Depending on the cadence ensure that enough time-travel or time-travel extension is available to ingest the stream. This latter point would only be applicable if the staleness goes beyond a defined time-travel period if it is less than the staleness extension period, read up more on this here: bit.ly/3eQqyAR. Time-travel is very handy in this regard because even if the landed content was truncated the time-travel partitions are still retrievable up until the time-travel period, up to 90 days as you have defined it.

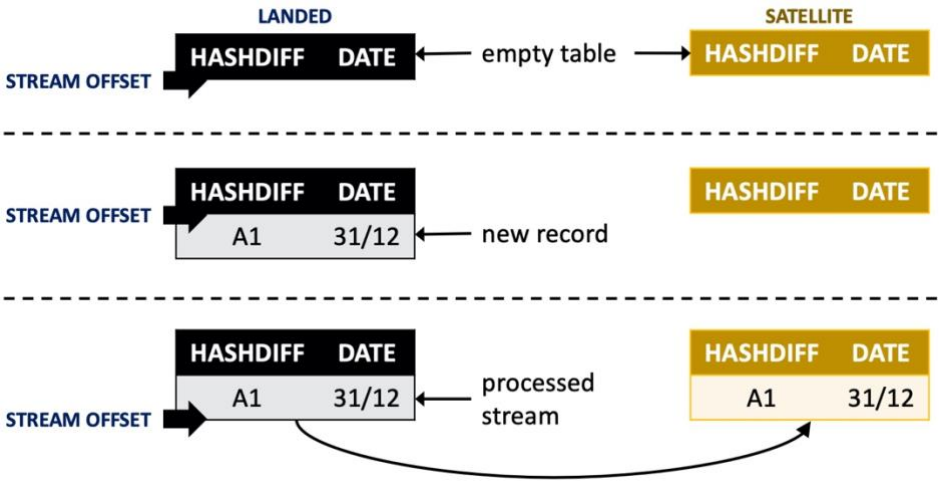


Figure 0-2 Basics of Snowflake Streams

STEP 2: Set up a Snowflake Task to only execute work when there is in fact **new** data in that stream, see: bit.ly/3JroZXW. This ensures that the task itself and child tasks will not execute unless data has been loaded. Tasks are set up using CRON syntax and therefore can scheduled to run multiple times in a day or week or another frequency you desire.

STEP 3: Modify staging code to deal with duplicates itself, *why it's nice to define Data Vault staging as a view!* Each state of a parent key will have a defined ***applied date***, if multiple loads occurred to the landing area *each package* of data will have a **different** applied date. By the time your satellite load code executes you could be handling multiple packages of data, the question is: *which are true changes?*

Add: "lag(*dv_hashdiff*) over
(partition by dv_hashkey_hub_account order by dv_applieddate) as *previous_dv_hashdiff"*
Add: "rank() over (partition by dv_hashkey_hub_account order by dv_applieddate) as dv_cnt"
Add: *"qualify dv_hashdiff <> previous_dv_hashdiff or previous_dv_hashdiff is null*

STEP 4: Modify satellite loading code to allow the second change through by default (because you have already determined it is a true change in staging) and the *first* change will be subject to the standard staged.hashdiff to target_satellite.hashdiff by surrogate hash key check. Staging must be cognizant of satellite splitting if you have applied them, *another reason to apply staging as views*.

Add: "or dv_cnt > 1"

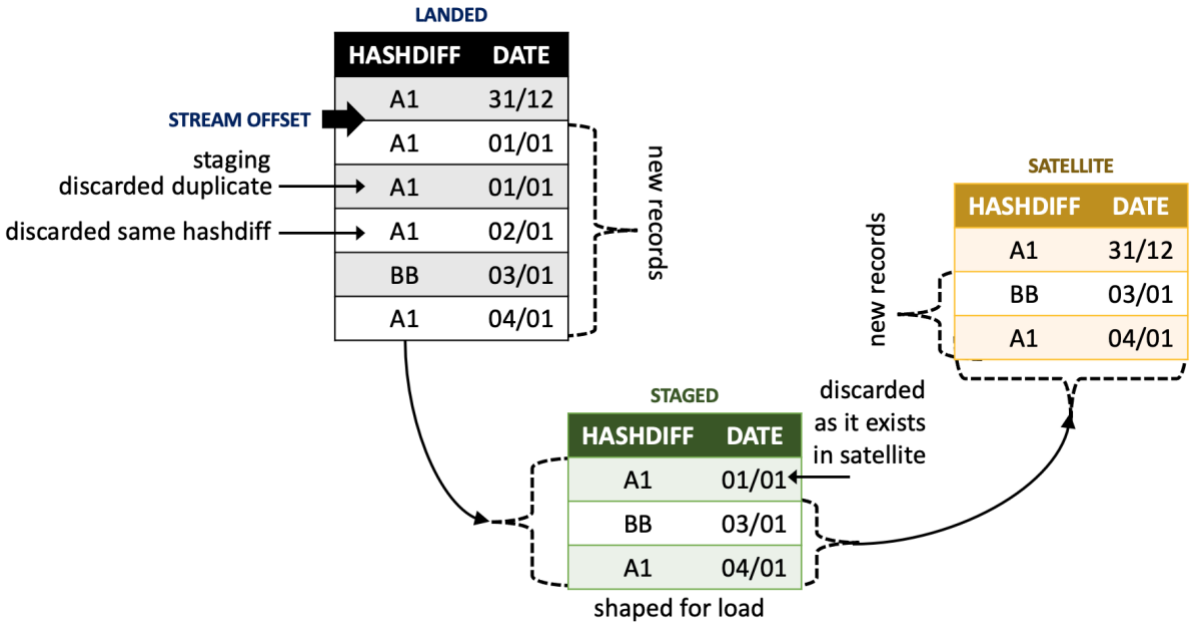


Figure 0-3 (single parent key example) Streams on Views would have the offset on the SQL View itself, diagram is to show what is new. Offset will natively "move" forward once DML is completed. Streams and offsets can form a part of explicit Snowflake transactions too

All complete code samples are available on the shelf! (or Kindle) amzn.to/3d7LsJV

These steps will work for Multi-Active Satellites, Effectivity Satellites, Record Tracking Satellites and Status Tracking Satellites but are not necessary for Hub or Link tables, after all they are unique lists in the target tables! The alternative for satellite table loading of course, is

if the source data are by definition always new, then we would use ***non-historized links*** *or* ***satellites*** *where change checking is not required!*

Retrieving the current record from a very big satellite table

The default answer to retrieving the current record from a satellite is to use an analytical function to dynamically *calculate* what the *current record* is and return the records that match that. Because Data Vault 2.0 is INSERT-ONLY no end dates and current flags exist on the satellite table. Retrieving 1.9 million current records from a 61 million record table using this method runs for 48 seconds, *not bad*. But maybe we can do better!

```
Current View:
select *
 from datavault.sat_card_masterfile
 qualify row_number() over
   (partition by dv_hashkey_hub_account  order by dv_applieddate desc, dv_loaddate desc) = 1;
```

If we had end dates and current flags, then that implies that we must perform SQL UPDATES on the satellite table when we load new records to it. An SQL UPDATE operation (as we know) creates micro-partitions and therefore a **costly operation** as it performs a copy of the original record row needing the update and persists it into the active table and the record as it was before the update is persisted to Time-Travel (inactive micro-partitions). One of the first items you check for improving query performance is the amount of **pruning** done on the underlying table. For an analytical function to determine what is the current record for a parent key it must scan *all* the underlying micro-partitions. No pruning is performed. An alternative to **static pruning** is **dynamic pruning**, and in Snowflake this is performed when an SQL JOIN executed. Yes… to get the current record we should be performing an SQL JOIN operation… *enter the current PIT tables,* bit.ly/3mNxuD9.

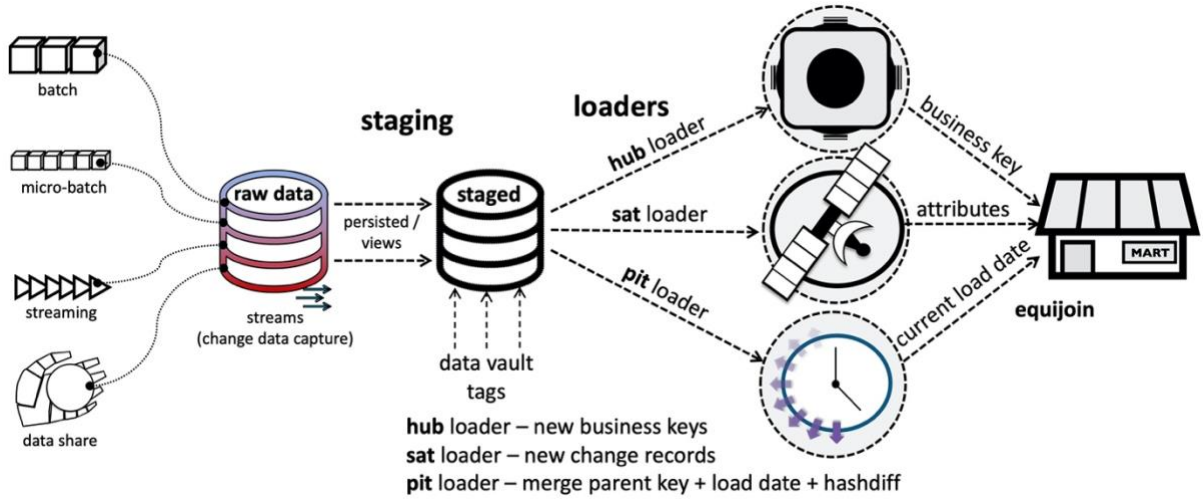


Figure 0-1 Another reason why EQUIJOINs matter! Satellite parent can be a hub or a link table

By maintaining a current PIT table that stores only the hash-key, the *latest* load date and **hashdiff** for that hash key, we use that table in every query that is only after the current active records of a table, *most reporting is looking for the current state anyway!*
The cost is minimal, because in essence an update to the current PIT table itself happens in parallel to updates to the satellite table and would probably complete earlier than the satellite

table update itself! The return on investment is that we can now retrieve 1.9 million records in 5.5 seconds from a 61 million record table.

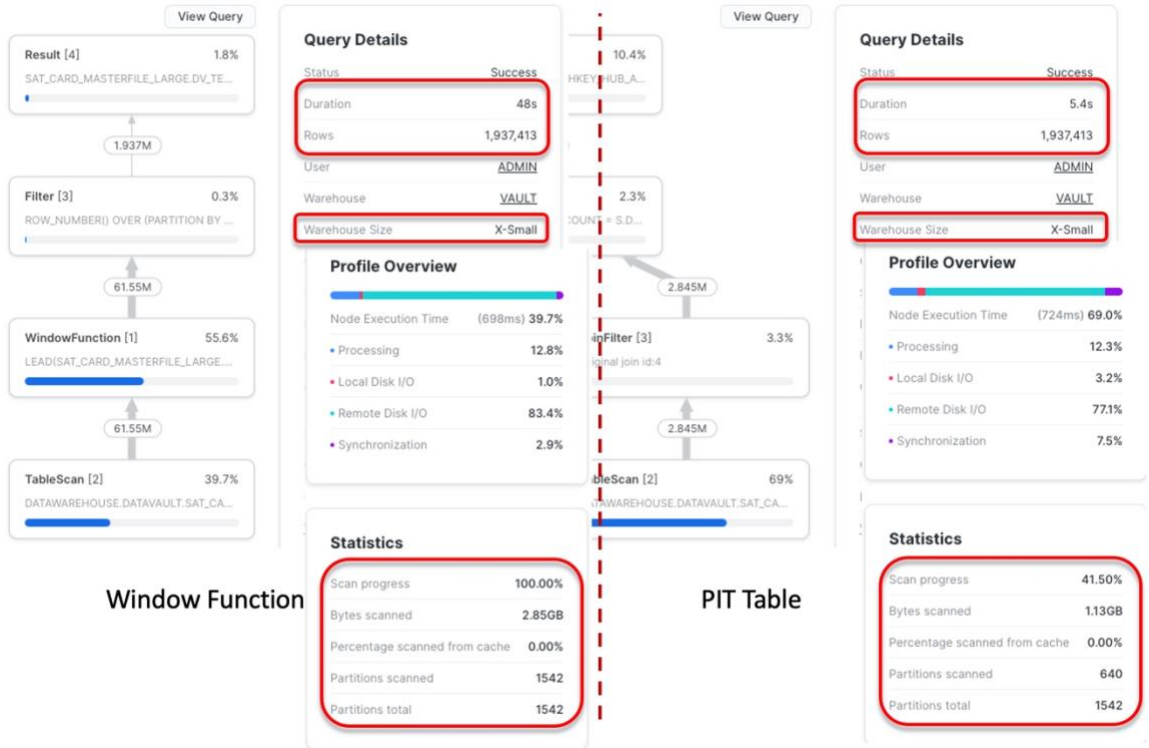


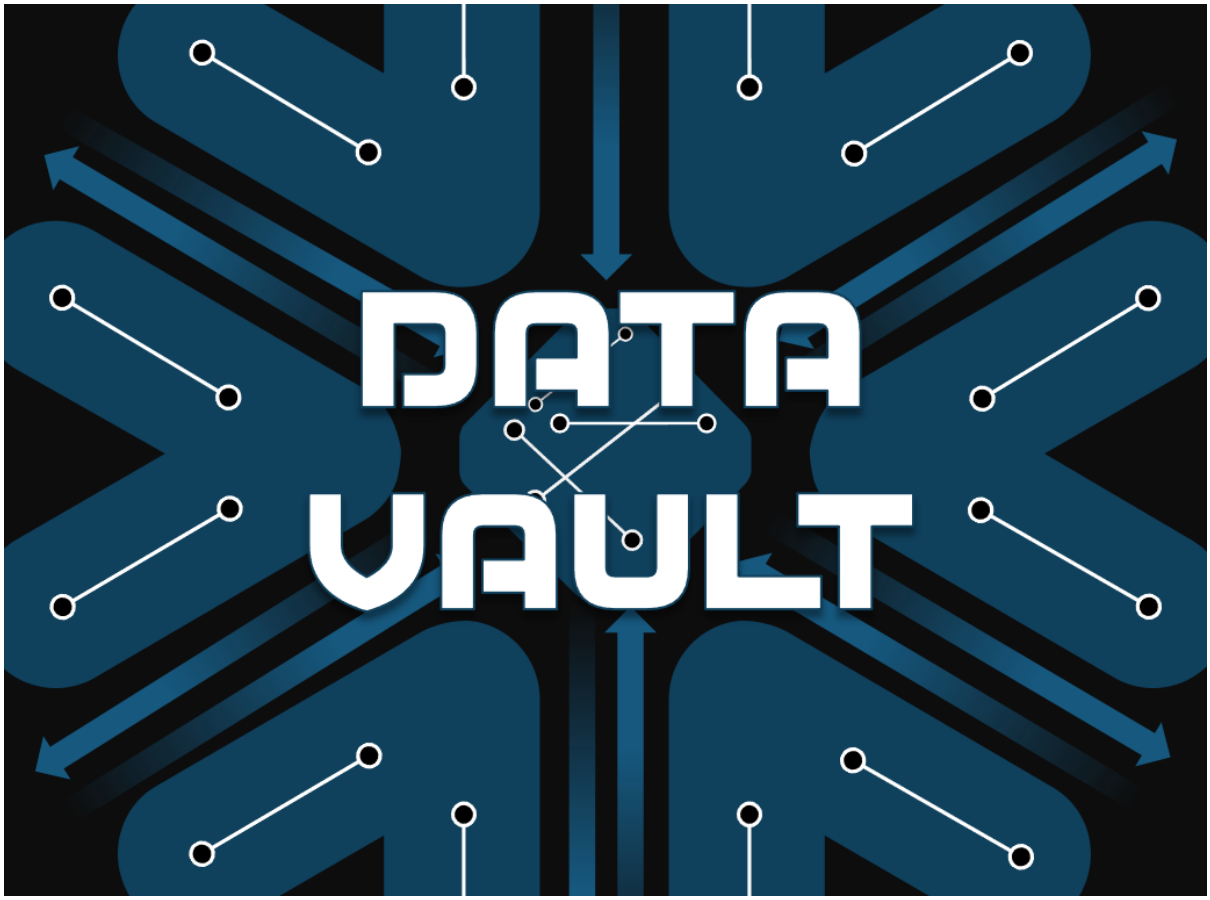
Figure 0-2 Comparing query profiles for returning current record, cache was flushed and result cache disabled

No need to cluster the underlying satellite table because the load order of new records ensure that the table is naturally clustered by the LOADDATE. If you were thinking about using the LAST_SEEN_DATE from the hub table as described above, well think about it. A hub table is *always* multi-source, a raw vault satellite table is not. The LAST_SEEN_DATE might not match a surrounding satellite table's LOADDATE. *Yet another reason why EQUIJOINs matter and why PITs are considered Query Assistance tables!*

In the end…

Yes, this article suggests some changes to how you *may* have built your data vault, you **must** understand the architecture you are working with and as such the implementation must yield to increasing pace of change (pun intended)! What I have shown in this article is that Snowflake does support this increased velocity and is as scalable as your Data Vault can be. Apologies if there are a lot of what might be esoteric concepts in this article as it digs into Snowflake ***and*** Data Vault! Always keep your ***cognitive load*** in mind! See: bit.ly/2ZYGpJP
*In Snowflake, SQL UPDATES and INSERTS **both** generate new micro-partitions*

Additionally, you can monitor which hubs are locked the most, Snowsight dashboard and code is here: bit.ly/3o8CC5i



#thedatamustflow #datavault #snowflake #snowsight

The views expressed in this article are that of my own, you should test implementation performance before committing to this implementation. The author provides no guarantees in this regard.