How can you be sure you load 100% of the data 100% of the time?

Loading all data, always. How does that work in practice? This is an article that expands on the application of Referential Integrity (RI) and consistency checks geared towards a Data Vault implementation. The concepts outlined in this article explain how to ensure consistency in your Data Vault from straightforward batch-based key-lookup approaches all the way down to applying consistency / RI checks in a fully independent parallel loading environment.

Eventual consistency

Referential Integrity (RI) is about managing consistency of data. It is the concept that relationships between tables in a database must always be (able to be) validated. For example, a foreign key value must exist as a primary key in the table that is referenced to by the foreign key (i.e. the 'parent' table) in order to satisfy RI.

RI is a concept that has been around for a long time, and it is a core design consideration for many 'data' solutions. In a typical DWH solution RI has traditionally been looked after both at the ETL process level as well as at the database level. This means that the ETL process, by virtue of key lookups, would assert whether the 'parent' key is available. If this is not the case the lookup would (should) return an exception.

To date this is still an acceptable practice. However, the introduction of hash keys (and natural business keys) add a layer of complexity (and opportunity) on the management of RI. Hash keys are an alternative implementation pattern to the use of meaningless keys and make it possible to load information in parallel and/or in a different order: an order that does not necessarily follow the table references. This, and the implications of independent parallel loading on managing consistency, is explained in more detail further in this article.

In addition to this, constraints can also be put in place at the database level to prevent non-consistent information to be entered: RI can be 'enforced' by the Relational Database Management System (RDBMS). Having said that, the most commonly agreed practice for this is to keep RI disabled at database level. The main reasons are the ever-increasing volumes of data and the fact that Data Vault solutions area able to span multiple platforms and technologies (i.e. different databases, partly on-cloud, partly on-premise etc.). Also, not every platform has the capabilities to RI constraints or transactional consistency.

Making sure RI validation is built-in when loading 'in parallel' using hash keys is still relatively easy to implement. Things get more challenging when you move towards a continuous loading mechanism where *every* process in the DWH solution runs *all the time*. If the solution is well developed it is possible to calculate the most recent point-in-time that RI *should* be OK and use this for validation, filtering and some powerful prioritisation mechanisms.

In a continuous loading environment, you seek *eventual consistency*. It is the acknowledgement that at the very present ('now'), the fringes of currently executing ETL processes, RI cannot be enforced - but that there will be a point in the not-too-distant future that everything (i.e. the 'lagging' ETL) catches up and RI can be validated again. In other words (inversely): with everything loading in parallel and near-real-time there is a point in time in the recent past that can be calculated and used to assert consistency. This value can then also act as filter criterion for consuming parties and solutions to prevent exposing 'incomplete' data.

Copyright 2018 Roelant Vos

One of the Data Vault mantras is '100% of the data 100% of the time' and validating consistency plays a large part in making sure this expectation can be met. Data Vault can cater for this, and there are elegant ways to implement this.

In this article I will explain how to apply RI on each of these approaches.

This article is written with a Data Vault based Data Warehouse (DWH) in mind, but arguably applies to many other DWH solution flavours as well.

Setting the scene – what a data delta means in the context of consistency

In practical terms, processing 100% of the data 100% of the time means that you can make sure you commit your entire received data delta to your DWH solution.

Data delta is the differential of changed records you receive (or pull) from your feeding ('source') systems. This can be implemented in numerous ways and is much harder than it is sometimes given credit for – but this is beyond the scope of the current article. In my Data Vault development & implementation training the source-to-staging concepts occupy at least a day because of the impacts these have on the overall design and patterns!

For the purpose of this article please assume we have Change Data Capture (CDC) interfaces available that load a set of inserted, updated and/or (logically) deleted rows into a dedicated Staging Area table. Once all data in the Staging Area table has been successfully committed to the DWH, the Staging Area table can be truncated and populated again with the next available set of data delta rows.

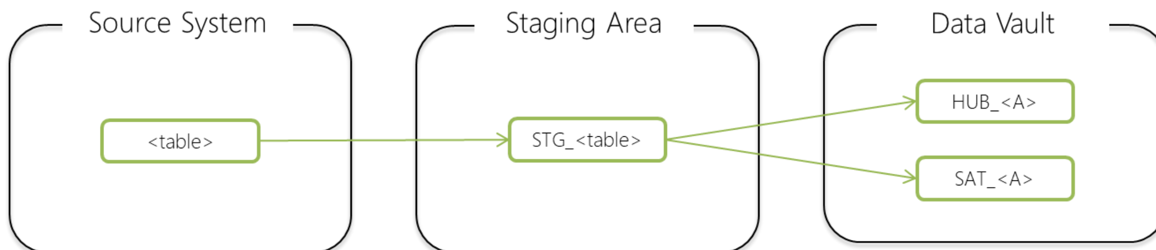
At this stage I need to clarify that CDC in this context is used in the narrow definition of detecting changes in values in source systems and being able to present these to the DWH. A purely technical 'tool' definition so to speak. This is in contrast to a broader perspective on CDC as ensuring only changes relative to the target (Data Vault) tables are loaded – something I refer to as a combination of 'change merging' and 'row condensing'. This is detailed at great lengths in the paper 'when is a change a change?' which explains how the Data Vault patterns can be configured to support system-level CDC. In summary, CDC in the context of this paper - to explain consistency concepts – is limited to the technical change detection on data sources.

This is the simplest way to look at consistency: only if all data has been properly committed the data delta can be removed and the next delta can be loaded. Or in other words - before you truncate the Staging Area you should ensure that RI is satisfied, that there is consistency. If you have developed your solution well, this should be the case and checks should be added to ensure this.

Copyright 2018 Roelant Vos

Some implementations don't use a Staging Area, but implement a Persistent Staging Area (PSA) which by definition contains all earlier data deltas (i.e. the full history). A PSA is never truncated, but appended with every data delta that comes through the system. The reason I bring this up is because at this stage I would like to clarify that, conceptually, a data delta in a Staging Area (which is transient / temporary in nature) is *the same* as a defined load window in the PSA.

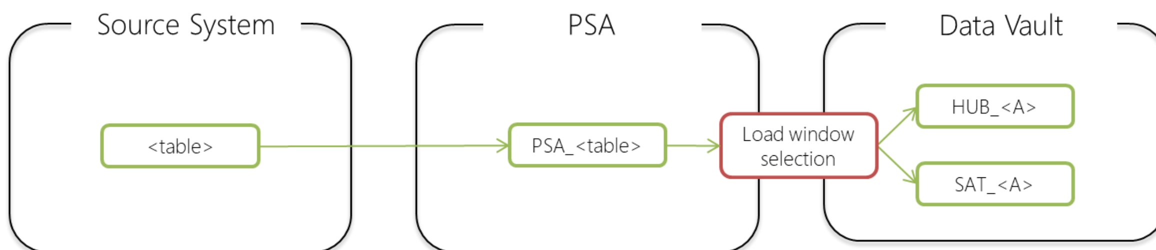
This is illustrated in the diagram below:



The data delta is applied to all mapped upstream tables in the Data Vault. Is this is completed successfully, the Staging Area table can be truncated.

The CDC interface detects changed records (the data delta). These are loaded into the Staging Area.

=



The data delta is applied as a load window selection on the PSA, and then loaded to all mapped upstream tables in the Data Vault.

The CDC interface detects changed records (the data delta). These are loaded into the Persistent Staging Area (PSA).

A load window is usually managed in a dedicated table as part of an ETL control framework, and can be as simple as the example below:

Load Window

Source Control Id	Process	Execution Date / Time	Load Window Start Date / Time	Load Window End Date / Time
1	A	2018-01-02	1900-01-01	2018-01-02
2	B	2018-01-03	1900-01-01	2018-01-03
3	C	2018-01-05	1900-01-01	2018-01-05
4	A	2018-06-06	2018-01-02	2018-06-06

Selection for process 'A'

Process	Load Window Start Date / Time	Load Window End Date / Time
A	1900-01-01	2018-01-02
A	2018-01-02	2018-06-06

The load window table captures the (history of the) range of data that is selected for a specific DWH load ETL execution. You can see that process 'A' is run at Execution Date / Time 2018-01-02 and has loaded a data set from the source PSA table covering 1900-01-01 up to and including 2018-01-02. This equates to the Source Control Id of '1'. The second time 'A' was run, on 2018-01-06 with Source Control Id '4', it loaded the data from (greater than) 2018-01-02 to (smaller or equal to) 2018-01-06.

The Load Window Start- and End Date / Time values used are based on the Load Date / Time Stamps (LDTS) from the PSA table. LDTS values are always sequential and managed by the DWH environment, and can therefore be relied on for auditing and filtering purposes (not for business interpretation).

Copyright 2018 Roelant Vos

Moreover, the load window is set at the *start* of the ETL process and includes both the start- and end-date. You are basically starting the process with a 'claim' on the row set you intend to process - a read lock if you like. This becomes more important later on in this paper.

Most ETL control frameworks have functionality for managing load windows, and the DIRECT framework as available on Github includes the table SOURCE_CONTROL for this purpose.

I appreciate there are other ways to load data into your DWH and a Staging Layer (Staging Area + Persistent Staging Area) in its entirety can be considered as an optional component of the solution architecture. Having said that, I believe the above helps to explain the more complex approaches and concepts later in the article. For now, the message is that a data delta in a Staging Area and a load window in the PSA are the same thing.

Let's look at some of the different patterns and approaches, and how to ensure consistency in these.

Looking after RI using key-lookups

Pros:

- Easy to implement, broadly understood
- RI constraints at database level possible
- RI contained in a single / individual ETL process
- Only requires the bare minimum ETL process control framework

Cons:

- Requires dependencies (workflow), which may impact scalability
- Upfront performance overhead due to lookup operation / caching

If you recall the discussions around the Natural Business Key, you may remember that Data Vault does not *prescribe* using hash keys. The Hub concept embodies the business concept, by virtue of the business key. Technically speaking the Hub entity maps the business key to a DWH key – a form of the key distribution concept.

It is still perfectly fine to consider using meaningless keys (such as integer sequences and identity columns) as implementation pattern for the key distribution, so let's start here as this is also the easiest way to explain how RI fits in. Implementing meaningless keys is a bit of a traditional approach, in the sense that it has been around for a long time and is generally well understood by the community. To avoid treading ground that is already covered I'll keep this brief.

Using meaningless keys forces you to implement dependencies in your workflow. You have to load the 'parent' table before the 'child' table. So, in order to load a data delta into a target Satellite table you would populate the corresponding Hub first so that the DWH keys are created (in the Hub) and therefore available for use in the Satellite.

When the Hub process is finished, the Satellite process can be run and will have to do a key lookup on the Hub (on the business key) to retrieve the meaningless key. If this is found then this (foreign) key can be inserted to the Satellite along with the rest of the attributes – and a reference has been established.

In this implementation pattern the ETL processes are the frontline to validate RI. If the lookup fails the process should throw an error as this means the key is not available in the Hub. You are basically preventing a RI violation. RI is enforced at *ETL level*.

Copyright 2018 Roelant Vos

This doubles as mechanism to ensure development standards have been properly followed (and unit- and SIT testing has been done) – that no shortcuts were implemented. When meaningless keys are implemented and the patterns have been implemented and orchestrated correctly there should never be a situation when data is inconsistent. If this does happen it is an indication that the loading processes are incomplete and / or design standards are not working properly

Because the ETL subsystem looks after RI in this manner, the RI constraints can also be enforced at database level (foreign key constraints). Depending on the technology you use this may help the optimiser for upstream queries, and if you don't want to enforce RI there is usually also an option to apply disabled constraints and still get the optimiser benefits. To be complete: RI constraints also come with a (varying / system dependent) performance overhead on insert. But we're not here to discuss database administration, we're focusing on the concepts.

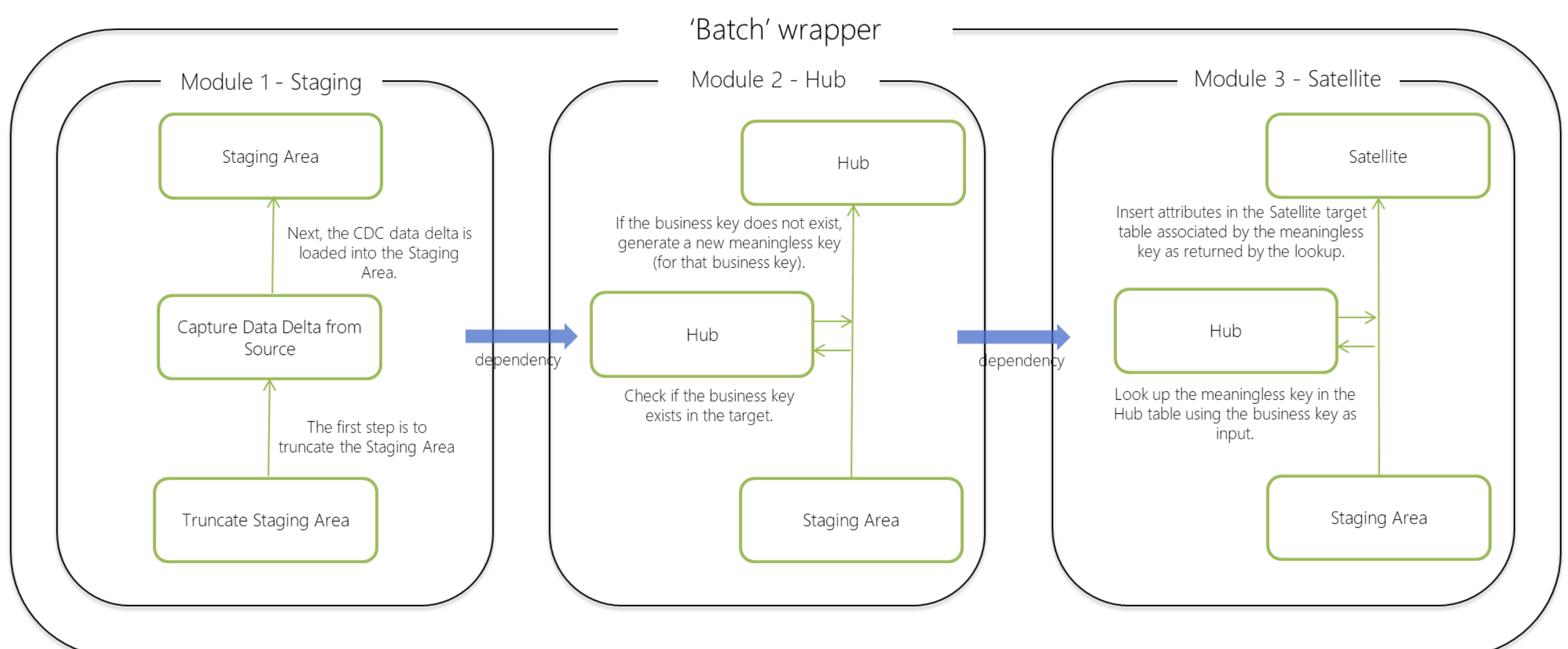
If you don't use the integer-style meaningless keys but prefer using hash keys or natural business keys you can still use the key-lookup pattern described here to enforce RI. This works in much the same way, although you only do the lookup to see if the key value *exists* in the Hub table (or not) - you don't need to actually retrieve it to commit the rows in the target table.

An example for the key-lookup approach is provided below to illustrate this pattern. Note that the terminology of the (open source) 'DIRECT' ETL process control framework has been used:

- A 'Module' is an atomic individual ETL process – an implementation of a pattern (i.e. a single Hub ETL)
- A 'Batch' is a collection of individual processes in a workflow or container for grouping and/or dependency management purposes.

A Batch is the standard executable / scheduled component because dependencies can be managed by the wrapper ('container') and don't need complex orchestration to be developed in a separate scheduler. However, Modules can also be executed independently which is something that is exploited in the fully independent parallel loading approach later in this article.

The example provided has defined a 'Batch wrapper' around three individual ETL processes (Staging, Hub and Satellite). When the Batch is started, it will first run Module 1 (the Staging Area ETL), then Module 2 (the Hub ETL) and last Module 3 (the Satellite ETL).



After all Modules have been completed the Batch is considered complete, which allows the Staging Area table to be truncated again. This can be done as a redundant final step in the Batch, or will otherwise be done automatically when the Batch is executed again.

You don't want to truncate the Staging Area delta before all ETLs that use it have been successfully run, and for this reason the Staging Area ETL step is preferably included as part of the Batch.

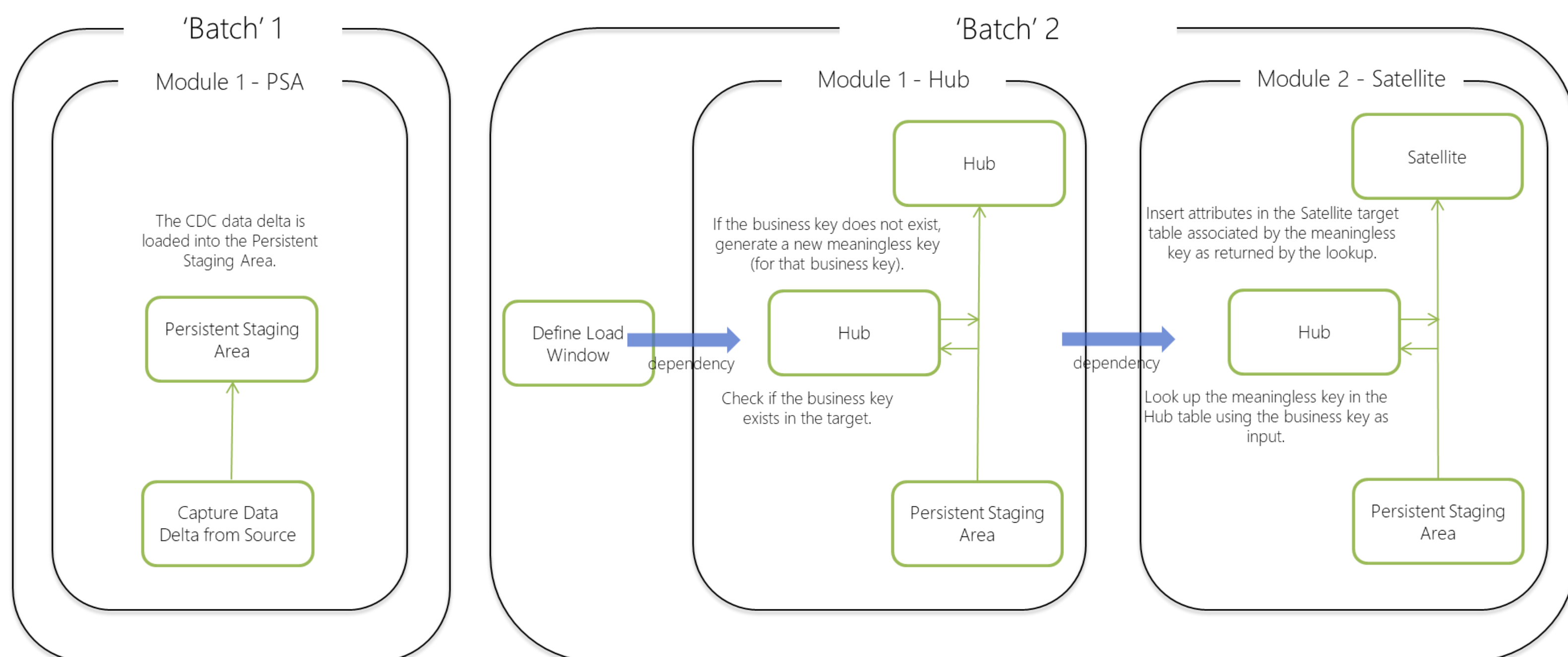
When the Batch has completed successfully, we should be able to assert that RI has been met.

Looking after RI using key-lookups, the PSA version with a load window

The key-lookup process basically operates the same when using a load window on the PSA. When you adopt a PSA-based approach the load window *should be set at Batch level* to ensure all processes work on the same data delta. As mentioned previously, having a (transient) Staging Area or a defined load window on the PSA is conceptually the same thing.

One of the benefits of this approach is that you can run the two Batches independently, the 'Staging' Module does not need to be part of the Batch as was the case in the previous section. The PSA batch can run separately from the Data Vault batch, and on different frequencies if required.

This decoupling is made possible by the load window. Consider the setup below:



In this approach the PSA table can be populated independently from the Data Vault loading processes. But, there is more to this. To allow this approach to work properly you need to be able to relate the data set that constitutes the load window (= the selection) with the execution status of the (instance of the) ETL process that populates the PSA table.

This is a bit abstract, but basically means that you have to know the status of the process that writes *to* the table you are selecting *from*. For example, we need to assert if data we want to select in the PSA table is still being written to. (i.e. has a status 'running' or 'completed').

Copyright 2018 Roelant Vos

Why is that? This is to make sure the upstream processes (i.e. the Hub, Satellite) only use a load window that covers data that is committed by the ETL process that loads data *into* the reference PSA table. The load window should not contain rows that are associated with an execution status that is 'running'. Or more precise: only rows that are associated with a process that has execution status 'successful' can be considered to be part of the load window.

The Load Date/Time Stamp (LDTS) is a good choice as attribute to be used to manage the load window, as it is incremental and fully controlled by the DWH solution. I often refer to this as the ETL control framework being able to apply 'ACID at ETL level' (as opposed to database level, by the RDBMS). You are basically preventing dirty reads, albeit at 'application level'.

It greatly helps if you can relate any record in your DWH with the unique execution instance of the process that touched the row – a best practice. Similar concepts are required for loading data into a Presentation Layer (from a Data Vault) as well.

Validating RI in (batch-based) parallel processing

Pros:

- Still a relatively easy solution
- Usually means a reduction of loading times due to parallel loading
- Usually means a reduction of loading times / performance gain due to removal of key lookup
- RI contained at workflow level as a sanity check, still easily manageable
- Does not conflict with 100% of the data 100% of the time: checks can be done *after* loading, not during

Cons:

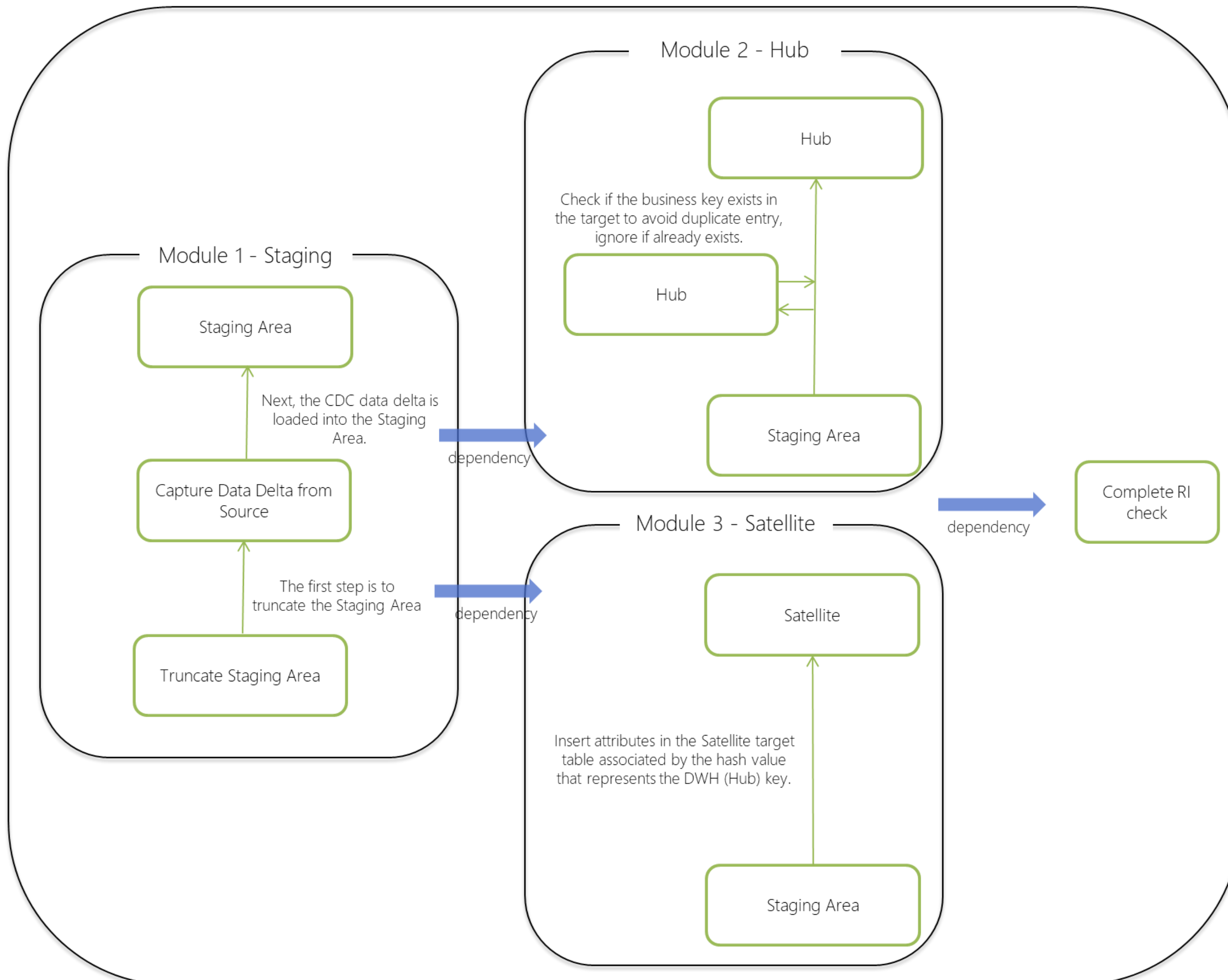
- Still has dependencies, because the 'batch' needs to complete as a whole before new data delta can be loaded to ensure consistency
- RI constraints at database level cannot be enabled. This is not necessarily a negative, more a consideration

To some, including myself, the prospect of parallel loading is very appealing and the adoption of hash keys makes it possible to load the Hub and Satellite independently, at the same time and/or in different order. The hash algorithm is deterministic and therefore you can rely (to a large extent – collision is out of scope here) that, as long as you use the same algorithm and business key, the hash value in the Hub will correspond with the (foreign key) equivalent in the Satellite.

Assuming you trust the hashing algorithm and have incorporated collision mitigation mechanisms (or get your business to accept the collision risk) you don't need to perform a key lookup at all and can therefore eliminate some of the 'cons' as listed in the key lookup section. The commonly accepted approach is to remove the key lookup in favour of relying on hashing with the associated rigor around using hash keys put in place, and this is how the example below has been created.

Copyright 2018 Roelant Vos

As we've seen in the previous section: the key-lookup 'performs' the RI check at ETL level. If you remove this function to allow for more parallelism, it is recommended to still implement this function at the 'batch' level or as a separate function. This is shown in the following diagram.



In this example, which uses a Staging Area, you can load any Data Vault ETL that uses the same Staging Area table in parallel. Only two are shown, but a typical Batch in this setup will contain 4-8 ETL processes. They can all run at the same time, but the Batch can only complete once all individual Modules have been completed and the RI check has run. Note that the Hub process still performs a 'lookup' but the purpose at this stage is only to prevent duplicate inserts ('check if exists, insert if not'). The Satellite process doesn't include a lookup because it relies on the hash key.

Copyright 2018 Roelant Vos

When adding the RI check you can limit the scope of records to the provided data delta, by linking to the Staging Area using a JOIN or EXISTS clause. Some examples are provided below:

```
SELECT COUNT(*) AS RI_ISSUES, ' for table <Satellite>'
FROM <Satellite> A
JOIN
(
  SELECT --hash algoritm i.e. MD5 (SQL Server example shown)
    HASHBYTES('MD5',
      ISNULL(RTRIM(CONVERT(NVARCHAR(100),<Business Key>)),'~N/A')+'|'
    ) AS <Hash Key>
  FROM
  (
    SELECT <Business Key>,
    FROM <Staging Area table>
  ) stgsub
) staging ON
A.<Hash Key> = staging.<Hash Key>
LEFT OUTER JOIN <Hub> B ON A.<Hash Key> = B.<Hash Key>
WHERE B.<Hash Key> IS NULL
```

If you would like to adjust the pattern there is also the option to move the RI check out of the Batch, and run this separately across the full DWH on times where there is a bit more capacity (i.e. after hours, the weekend). It goes without saying this is an expensive query and may not be scalable in your environment.

```
SELECT COUNT(*) AS RI_ISSUES, '<Satellite>'
FROM <Satellite> A
LEFT OUTER JOIN <Hub> B ON A.<Hash Key> = B.<Hash Key>
WHERE B.<Hash Key> IS NULL
```

And a Link example that connects three Hubs together:

```
SELECT COUNT(*) AS RI_ISSUES, ' for table <Link>'
FROM <Link>
LEFT OUTER JOIN <HUB_1> ON <Link>.<Hub 1 Hash Key> = <HUB_1>.<Hash Key>
LEFT OUTER JOIN <HUB_2> ON <Link>.<Hub 2 Hash Key> = <HUB_2>.<Hash Key>
LEFT OUTER JOIN <HUB_3> ON <Link>.<Hub 3 Hash Key> = <HUB_3>.<Hash Key>
WHERE (
  <HUB_1>.<Hash Key> IS NULL OR
  <HUB_2>.<Hash Key> IS NULL OR
  <HUB_3>.<Hash Key> IS NULL
)
```

These last two queries look a bit simpler, and can rely on the model references and are therefore easy to generate. However, the good news is that you can generate *all* of the above checks (including those that join to the Staging Area) with the exact same metadata as you used to generate the ETLs and workflows themselves.

Everything can be generated from metadata. This makes sense – because you need to understand the relationships these tables have with each other and their sources and targets (the 'edges of the graph'). More information on how to achieve this is part of the various ETL generation and automation topics and code (i.e. TEAM, DIRECT and VEDW). The VEDW tool has a function to generate both these scripts based off the generation metadata.

This is an effective approach to develop a micro-batch solution. Every Batch is essentially independent and makes sure it maintains consistency. This makes it easy to increase or decrease the frequency of execution without changing the design. And, similar to the key-lookup example, the Staging Area table can be replaced with a PSA Load Window in exactly the same way which provides another level of independency for scheduling due to the load window decoupling.

Copyright 2018 Roelant Vos

Eventual consistency: asserting RI when everything runs continuously and independently

Pros:

- Close to near real-time approach, data can be loaded immediately / as soon as it is available
- Usually means a reduction of loading times due to parallel loading
- Usually means a reduction of loading times / performance gain due to removal of key lookup
- Does not conflict with 100% of the data 100% of the time: checks are done after loading, not before
- No ETL dependencies whatsoever

Cons:

- Additional complexity in orchestration and calculating when RI should be satisfied (point-in-time)
- Requires an ETL control framework. Having a control framework is considered best practice in my view, but previous approaches can theoretically be implemented without one. For eventual consistency however, a control framework is *mandatory*.
- RI constraints at database level cannot be enabled. This is not necessarily a negative, more a consideration
- Requires a direct-to-Data-Vault, PSA or Staging Area approach that retains data until all data has been committed, which usually means retaining multiple data deltas for a period of time. This is not necessarily a negative, more a consideration.

A more sophisticated way to load data is on a continuous schedule, and the extreme implementation of this discards the 'Batch' concept altogether. No longer is a data delta maintained until all upstream processes have completed. Instead, all individual processes are executed in any order as directed by a queuing mechanism. Priority can be driven through any set of parameters and/or system inputs - and moves the solution closer towards the self-optimising 'engine' configuration.

In this continuous loading (queue) environment the Hub and Satellite processes used in the earlier examples no longer run as part of a Batch, which is another way of saying that they no longer have a dependency built-in to process the exact same data delta from a Staging table. Instead, in the continuous loading environment each process has its own intervals to process - its own individual data delta for their PSA table. This is explained in more detail in the paper about eventual consistency.

For all the purposes of explaining the concepts, the continuous loading approach as outlined in this section requires the Data Vault to source its data from a PSA, or at least a Staging Area that persists the data delta until *all* data has been committed. In a parallel continuous loading approach this requires the Staging Area to have clean-up functionality that removes 'old' load windows as opposed to a truncation of the entire table. For the purpose of explaining the concept I will use the PSA.

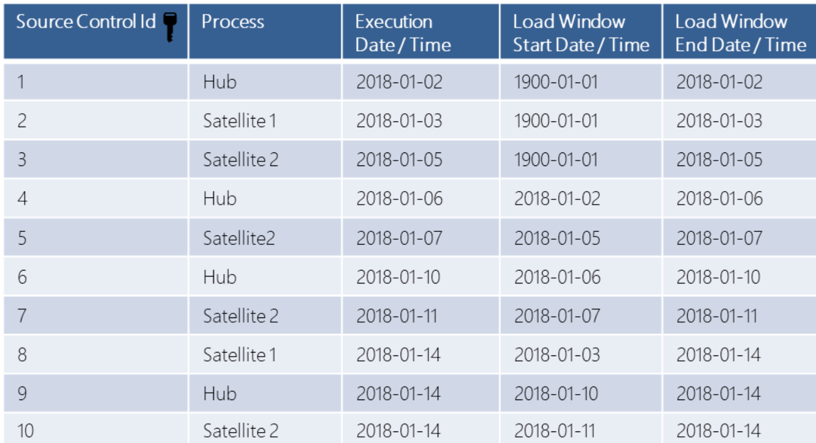
In practice this means that, for example, a Satellite may run three times before the corresponding Hub is loaded. Consequently, RI is only satisfied after the Hub has caught up.

But it is more complicated than that. The Hub in this example could have a load window may have reached further than the last (of the three) load windows related to the runs of the Satellite process. So, we need to take these timelines into account in order to assert RI, and acknowledge that this slightly lagging in time (of execution). Theoretically, we can validate RI across the solution by stopping all processes and make sure every ETL process has caught up before running the checks – but the challenge is to be able to do this while everything keeps running.

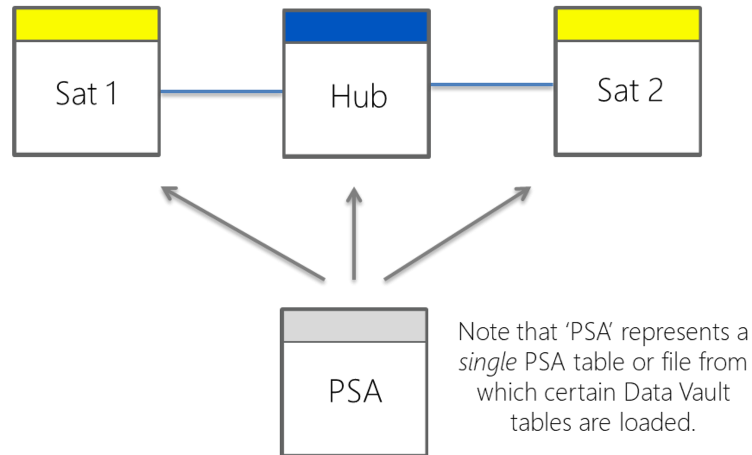
So, how to detect the most recent point in time when referential integrity (RI) *should* be met in the context of continuous loading?

Copyright 2018 Roelant Vos

To create a scenario to work with, consider the sample table below that contains the load windows of three individual processes, a Hub and two related Satellites. To keep things simple these three processes load from a single PSA table (or file). In the 'real world' many Data Vault objects are likely to be populated from many different sources (different PSA tables in the context of this example), but this does not change the concept or implementation.


Load Window

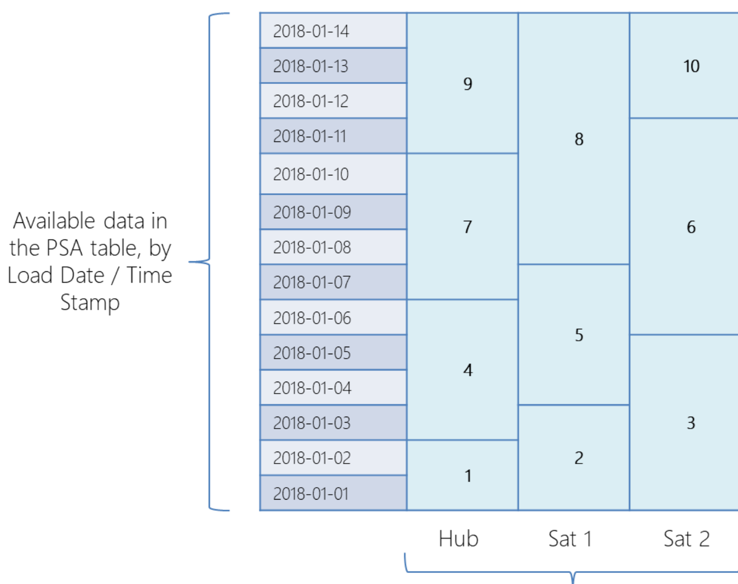
Source Control Id	Process	Execution Date / Time	Load Window Start Date / Time	Load Window End Date / Time
1	Hub	2018-01-02	1900-01-01	2018-01-02
2	Satellite 1	2018-01-03	1900-01-01	2018-01-03
3	Satellite 2	2018-01-05	1900-01-01	2018-01-05
4	Hub	2018-01-06	2018-01-02	2018-01-06
5	Satellite2	2018-01-07	2018-01-05	2018-01-07
6	Hub	2018-01-10	2018-01-06	2018-01-10
7	Satellite 2	2018-01-11	2018-01-07	2018-01-11
8	Satellite 1	2018-01-14	2018-01-03	2018-01-14
9	Hub	2018-01-14	2018-01-10	2018-01-14
10	Satellite 2	2018-01-14	2018-01-11	2018-01-14



Remember: the load window (the data delta) is based off the Load Date / Time Stamp (LDTS) as is available in the PSA. This means that based on the above table you can see that on 2018-01-02 the Hub was loaded using all available data between 1900-01-01 and 2018-02-01. In reality, this comes down to micro seconds but for the purpose of explaining the concepts I use the 'day level' abstraction.

The question we seek to answer is: has a specific data delta been committed to the DWH? In other words: have all ETLs that load from the same PSA have an overlap?

The easiest way to approach this is to map out the overlaps in the load windows of the involved processes. This results in the following diagram:



Individual ETL process executions, per process. The number corresponds with the 'Source Control Id' in the Load Window table.

How does this help? By looking at the load windows per process, we can see that the first time that all processes have loaded *the same set of overlapping data delta* is on 2018-01-02.

This is when the ETL executions 1, 2 and 3 have completed successfully, but even though process 3 has loaded data up to and including 2018-01-05 RI can only be asserted for data up to 2018-01-02.

Why only up to 2018-01-02? This is because process 3 is associated to the execution of the 'Satellite 2' ETL process, which for the period 2018-01-03 to 2018-01-05 may have no corresponding Hub keys. This is because the Hub ETL hasn't been able to process this range of data as of yet, there is no overlap yet.

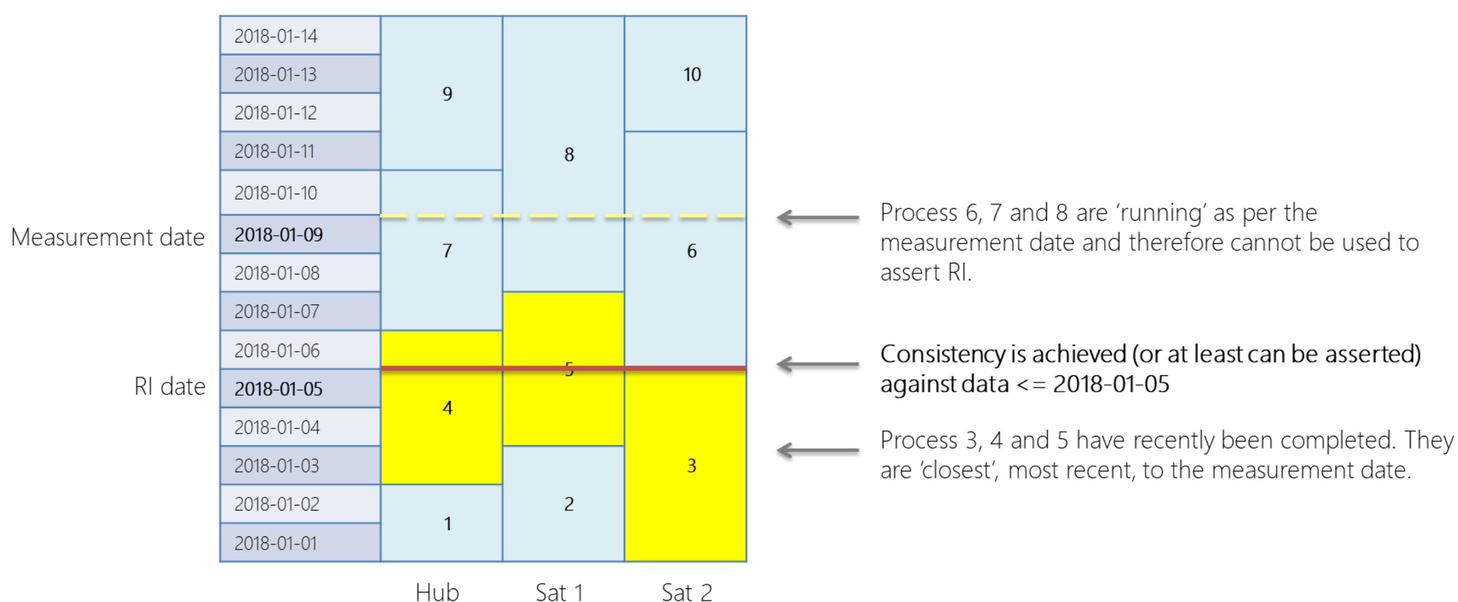
Copyright 2018 Roelant Vos

Now let's say we would like to simulate (calculate) the most recent RI point-in-time for a specific date, say 2018-01-09. At this point in time process 6, 8 and 7 are running and therefore cannot be used to assert RI. Process 9 and 10 haven't run as of yet. At the same time we know process 1 – 5 have been completed.

To find the answer we're looking for we need to know which processes have been (successfully) completed, and what data deltas were associated with these processes. If we know these delta ranges we can understand what the most recent point in time was that there was 'overlap' and therefore consistency. Remember: data deltas are all individual and tied to a unique ETL process, and therefore very likely different!

We're not interested in processes that had a subsequent successful run relative to the measurement date (2018-01-09), we're only interested in the most recent ones. This means we can exclude execution instance 1 and 2 as they had a new completed process relative before 2018-01-09 (4 and 5 respectively).

So – we will need to understand the *most recent successful executions* that use the same source and calculate the most recent point of overlap. This is the end of the load window from the process that finished first, simply because the other load windows captured 'more data'. This is explained in the next diagram:



This diagram shows that if we measure at 2018-01-09, the most recent executions are 3, 4 and 5 and the minimum load window end date / time is 2018-01-05. We should only select data from our Data Vault smaller or equal to this date, because any more recent data is not consistent yet.

To summarise: in order to validate at what point in time RI can be assessed the following steps can be used:

1. Using a measurement date as input, retrieve the *most recent* (successfully) completed ETL processes relative to the input date
2. Select the *minimum* (lowest) Load Window End Date / Time from the result of step 1. This is the date at which RI should be correct.

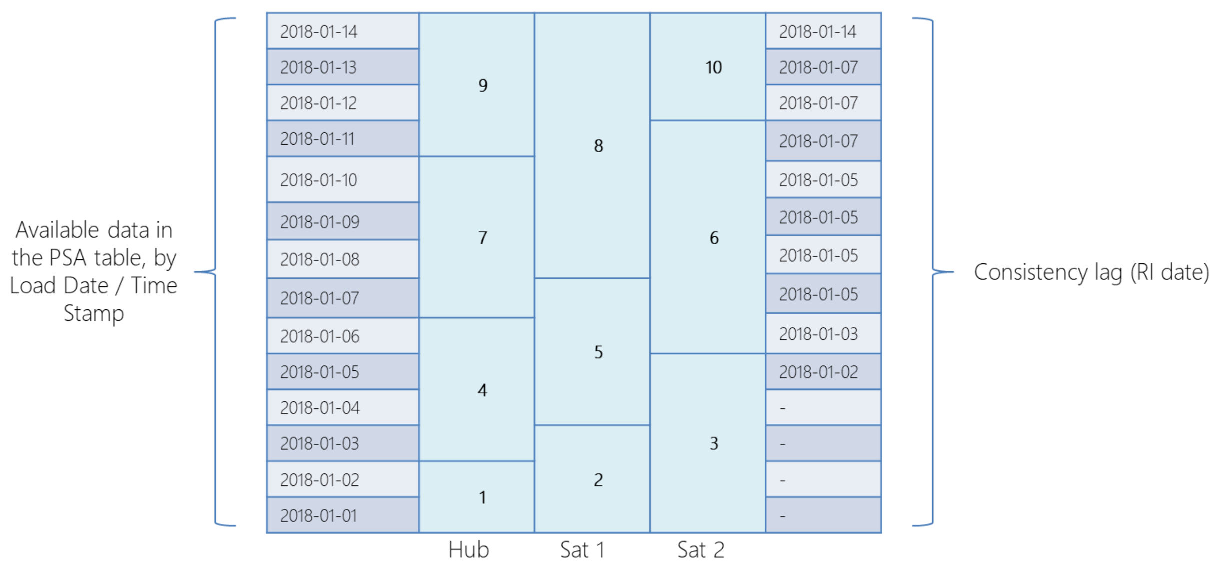
Why are we talking about selecting *from* the Data Vault at this stage? This is because we only should seek to expose data to upstream consumers (information marts, services) that can be considered consistent in the continuous independent loading. We know that eventually the environment will be consistent – and with the above logic we can calculate this point-in-time and use this as a filter to consuming areas. This is a powerful mechanism which provides both the flexibility of full parallel loading with the functionality to prevent exposing incorrect data.

Copyright 2018 Roelant Vos

As measurement date you typically would use 'now' (i.e. system date/time). As quickly as you receive the data you may want to push this to an information mart, so you typically would ask the system what is available 'now'. But, the logic works as well for a 'current' date / time as it would for a historical date / time, the latter being useful for troubleshooting RI issues that may appear due to development shortcuts.

It is worthwhile to explain what 'date' we are actually referring to here. So far, we have used the PSA LDTS to create the load windows and when you load data from the Data Vault to an Information Mart this may not seem logical to use as a filter. However, remember that in this approach we *inherit* the Staging Area or PSA LDST to the Data Vault, so it will directly relate to the LDTS (effective date / time) in the Satellite.

In my view, this is how the LDTS needs to be used: as a technical filtering mechanism for consuming areas. This is shown here in the next diagram:



If, say, there hasn't been any new data in my PSA table and the present is 2018-08-10 the formula will return 2018-01-14 as the most recent date for consistency – which equates to 'everything'. But you don't have to worry about this date, all this logic can be wrapped up in a function that can be used as filter criterion in any selection.

Assuming the default measurement date is 'now' you simply select from a Data Vault where the LDTS / Effective Date is smaller or equal to the value the function returns. Examples of how to do this are available in the next section.

The above diagram also displays the lagging of the consistency in the near real-time environment. The more you increase the queue frequency, the closer the consistency date is to the present. If you decrease the frequency you can expect this gap to increase.

The ability to calculate the most recent point of consistency also provides a set of interesting new metrics for the priority setting of the queue. For instance, you can now also prioritise on the impact an individual process has on lagging the RI high water mark and 'bump' its priority.

This makes for an awesome Key Performance Indicator (KPI) for the management of the DWH environment, and can be used to inform the business users and other consuming parties about timeliness and readiness. It's a powerful approach.

Copyright 2018 Roelant Vos

Implementation guidelines for eventual consistency

There are many ways to implement this logic, and I have added two approaches to consider. The first one, which is the easiest, is to calculate consistency based on a series of related ETL processes. This is similar to the example used earlier in this article where you have a Hub and two Satellites loading from the same PSA table.

Using the metadata that is used to generate them in the first place, you can derive that these three ETL processes are related by virtue of sharing the same source. A very straightforward way to wrap this in a function is included below. This uses XQuery to transpose an array into a list that can be used to retrieve the minimum Load Window End Date / Time from the most recent successful ETL executions.

```
DECLARE @MeasurementDate datetime2(7) = GETDATE() -- For example '2018-01-09', or GETDATE()
DECLARE @ConsistencyDate datetime2(7)

DECLARE @ETL_Process_Array VARCHAR(MAX) =   'Hub, Sat 1, Sat 2' -- A sample list of related ETL processes

-- Capture the array in XML for transposing later
DECLARE @ETL_Process_Array_XML XML = CONVERT(XML, '<a>'+replace(@ETL_Process_Array,',' ,'</a><a>')+'</a>')

-- Calculate the output
SELECT @ConsistencyDate = MIN(<Load Window End Date/Time>)
FROM (
SELECT
       <Source Control ID>,
       <Process Execution ID>,
       <Load Window Start Date/Time>,
       <Load Window End Date/Time>,
       ROW_NUMBER() OVER (PARTITION BY COALESCE(<Process ID>,0) ORDER BY <Process Execution ID> DESC) AS ROW_ORDER
FROM <Load Window Table>
WHERE
      <Execution Status Code> = <Successful>
  AND <Load Window End Date/Time> <= @MeasurementDate
  AND <Process ID> IN
  (
    SELECT <Process ID> FROM <ETL Process Register>
    WHERE <Process ID> IN (SELECT A.value('.', 'varchar(max)')  FROM @ETL_Process_Array_XML.nodes('a') AS FN(A))
  )
) sub
WHERE ROW_ORDER = 1 -- Get the most recent successful execution

PRINT @ConsistencyDate
```

This will return back the most recent consistency date for the list of ETL processes that was provided as input.

Ideally though, you would want to ‘turn this around’ and start with the Data Vault table in mind. We can extend the logic above to support providing a series of Data Vault tables as input. This is the ultimate way to implement this logic, and can scale to cover a subject area (i.e. a business concept) or the entire DWH.

Copyright 2018 Roelant Vos

The example below uses a Common Table Expression (CTE) to replace XQuery, which is there to show different ways to achieve the same thing. This approach also needs access to metadata which links an ETL process to a (target) table, and therefore places additional requirements on your ETL control framework metadata.

```
DECLARE @MeasurementDate datetime2(7) = GETDATE() -- For example '2018-01-09', or GETDATE()
DECLARE @ConsistencyDate datetime2(7)

DECLARE @Table_List VARCHAR(MAX) =   'Hub, Sat 1'; -- For example 'HUB_CUSTOMER'

WITH myCTE(start_position, end_position)
AS
(
SELECT CAST(1 AS INT), CHARINDEX(',', @Table_List)
UNION ALL
SELECT end_position + 1, CHARINDEX(',', @Table_List, end_position + 1)
FROM myCTE
WHERE end_position > 0
)
, tableArray AS
(
SELECT DISTINCT DATA_STORE_CODE =
SUBSTRING(@Table_List, start_position,
          CASE WHEN end_position = 0 THEN LEN(@Table_List)
               ELSE end_position - start_position
          END)
FROM myCTE
)
SELECT @Output = MIN(<Load Window End Date/Time>) -- Calculate the output from the CTE
FROM
(
SELECT
       <Source Control ID>,
       <Process Execution ID>,
       <Load Window Start Date/Time>,
       <Load Window End Date/Time>,
       ROW_NUMBER() OVER (PARTITION BY COALESCE(<Process ID>,0) ORDER BY <Process Execution ID> DESC) AS ROW_ORDER
FROM <Load Window Table>
WHERE
      <Execution Status Code> = <Successful>
  AND <Load Window End Date/Time> <= @MeasurementDate
  AND <Process ID> IN
  (
    SELECT <Process ID>
    FROM <Table Registry>
    INNER JOIN tableArray ON tableArray.<Table Name> = <Table Registry>.<Table Name>
    INNER JOIN <ETL / table xref> ON <ETL / table xref>.<Table Name> = <Table Registry>.<Table Name>
  )
) sub
WHERE ROW_ORDER = 1 -- Get the most recent successful execution

PRINT @ConsistencyDate
```

A final note on this implementation example is that it may be required to filter out ETL processes / sources that haven't been refreshed in a while. For example, for a reference table that is only rarely updated you can expect the most recent information to be some time in the past – and this would be correct because there haven't been any changes. This needs to be catered for, otherwise this will manifest itself as a lagging process for no other reason that no new load windows have been created (because there was no data to process).

In short, you seek to filter out ETL processes that have been run successfully but didn't pick up any data delta because there wasn't any available to process.

Copyright 2018 Roelant Vos

The DIRECT framework creates a load window entry for every execution, which is essentially redundant for PSA tables that are not appended to regularly. For example, this would create a Load Window Start Date / Time which is the same as the Load Window End Date / Time. We can use this mechanism to remove ETL instances which correctly don't have data to contribute to the consistency check and thus eliminate this edge case. This is one way to achieve the desired outcome.

Should, or is?

This paper has outlined various techniques and approaches on asserting DWH consistency including RI checks at ETL level, constraints at RDBMS level and various (SQL) validation and detection queries. This variety of content highlights a difference between 'hard' RI and 'soft' RI, or in other words that some checks will *prevent* consistency issues where others are meant to *alert* on consistency issues.

This is especially relevant in the eventual consistency approach: we can calculate at which point in time consistency *should* be in order, but this in itself doesn't necessarily mean it *is*.

This is why it is worthwhile to consider functionality in the ETL control framework that captures the points in time where the RI has been validated, for instance by using the queries provided in the 'Validating RI in (batch-based) parallel processing' section. If you have calculated RI at various points and keep a record of this you can reduce performance impacts of the RI validation queries by limiting the date ranges. Also, you can use this as a filter for upstream data processing (i.e. to an Information Mart).

Following the LDTS timeline, and using this as filtering mechanism the Data Vault is considered insert-only. So, when RI has been asserted at a certain date / time this will not change again.

Final thoughts

This article explains how to ensure consistency in various scenarios, but it would not be correct to assume it covers every combination of concept and technology. As I made note of in the introduction of the article, in some solution architectures there is no Staging Layer at all.

This is especially true in the world of true near-real-time parallel loading at scale where transactions don't 'stop' in a (Persistent) Staging Area but are loaded directly into the Data Vault. In a message queue style of direct-to-Data-Vault loading there is no such thing as a load window. Therefore, these designs require additional considerations on CDC (in both the narrow and broad definition) and monitoring of associated load windows ('consistency windows').

As always – it is all about being aware of your design and selection of associated patterns to develop a fit-for-purpose solution: options and considerations.

At implementation level, bear in mind that the required queries to calculate the consistency levels can have a significant impact on your ETL control framework subsystem. Please investigate and test this for potential performance impacts. A result may be that continuous reads on the control framework may create read locks on the database that prevents newly spawned processes from executing (because they have to write to the framework).

One solution is to carefully manage transactions and isolation levels, for instance by allowing the server to apply optimistic locking approaches (i.e. snapshot isolation / row versioning). Also, consistency checks need to consider NULL values in source systems. Last, but not least, be mindful that not all relationships will be available. In some cases, there is no other option to map a relationship to an unknown key (dummy key) in a Hub.

Don't forget to create those, otherwise the RI checks will remind you!

Copyright 2018 Roelant Vos