Merging history

Beyond creating Hubs, Links and Satellites and current-state (Type 1) views off the Data Vault, one of the most common requirements is the ability to represent a complete history of changes for a specific business entity (Hub, Link or groups of those). If a given Hub has on average 3 or 4 Satellites, is it useful at the very least to see the full history of changes for that specific Hub across all Satellites.

How to merge various historised (time-variant) data sources such as Satellites together, especially when you have more than two of them, essentially reuses some concepts that have been around for many years and in many forms. They can generally be referred to as 'gaps and islands' of time (validity) periods.

In this post I intend to explain how this concept can be applied to Data Vault to create both Point-In-Time (PIT) tables as well as Dimensions. The Dimension in this example would contain the full history for every attribute (completely 'Type 2' – in Dimensional Modelling) but of course can contain a mix at attribute level as to how history is presented. Ultimately it is a consideration for every individual attribute how this is done, but for the context of this post I will assume the entire Dimension requires the historical / Type 2 / time-variant perspective.

An additional disclaimer is needed here for PIT tables as well: if you have read any of the Data Vault books you may notice that my interpretation of a PIT table is slightly different. The explanations in the books is geared towards storing and maintaining snapshots of 'state' for a group for various intervals (e.g. days as per midnight or something like that). This is a very sensible way to define this, but in this post I will focus more on combining the history at the most detailed level. The reasoning for this is that a 'state per interval' can always be derived from the full history and using the same techniques outlined in this post. In other words, from the full history you can present the point-in-time snapshot as outlined in Dan Linstedt's book very easily.

The default PIT table is a combination of the various Satellite keys and date attributes against the lowest defined grain (the entity, Hub), but it is perfectly OK to add attributes or even business logic if it makes sense for you. In the end, these are performance measures that need to be fit-for-purpose. The more attributes you add, the more the raw result starts to look like a Type 2 (full history) Dimension.

To keep some differentiation between the two let's agree that a PIT table combines history from its direct surrounding tables (all Satellites for a Hub for example). A Dimension is broader than this and can contain more historised sets (e.g. multiple Hubs, Links and their Satellites). If in any way possible, I always try to generate the Dimensional Model directly off the Data Vault and only use PIT tables if there is really a performance issue. As a result, I don't really use PIT structures that much. But if you need that bit of additional performance loading a Dimension they do come in handy.

As per the above introduction, the difference between PIT tables and Dimensions isn't really big and the logic to create both is therefore very similar. The main difference is that creating a Dimension consisting of multiple sets of Hubs, Links and Satellites requires some creativity around navigating through the model. Which path to take, preventing cycling back to tables that are already incorporate – that sort of thing. But once you have mapped your path through the model the logic to put it all together is the same.

I have applied these concepts in the Virtual EDW software so I can easily 'play around' with various structures and in terms of generation / automation of this I can attest the logic is pretty much the same indeed: once you have the tables and paths you're basically done. Feel free to download the latest version and try this out yourself, but let's start with the basics first.

Joining time-variant data sets together

The basic approach to join two time-variant tables together is to join them on their shared key (CUSTOMER_SK in the example below) as well as their Effective and Expiry Date/Times. This is -by far- the easiest if you have an expiry date/time, of course, but if you don't want to persist this you can always derive it off the effective date/time. In the example below I persisted the expiry date/time in the available underlying data.

The join logic is essentially making sure the overlap in time-periods (I always refer to this as time slices) is evaluated: the greatest of the two effective dates is smaller than the smallest of the two expiry dates. Everything else we do is a variation of this theme. Consider the example below:

```
SELECT
  A.CUSTOMER_SK,
  (CASE
    WHEN B.OMD_EFFECTIVE_DATETIME > C.OMD_EFFECTIVE_DATETIME
    THEN B.OMD_EFFECTIVE_DATETIME
    ELSE C.OMD_EFFECTIVE_DATETIME
  END) AS OMD_EFFECTIVE_DATETIME, -- greatest of the two effective dates
  (CASE
    WHEN B.OMD_EXPIRY_DATETIME < C.OMD_EXPIRY_DATETIME
    THEN B.OMD_EXPIRY_DATETIME
    ELSE C.OMD_EXPIRY_DATETIME
   END) AS OMD_EXPIRY_DATETIME, -- smallest of the two expiry dates
  * -- select attributes
FROM HUB_CUSTOMER A
JOIN SAT_CUSTOMER B ON A.CUSTOMER_SK=B.CUSTOMER_SK
JOIN SAT_CUSTOMER_FAST_CHANGING C ON A.CUSTOMER_SK=C.CUSTOMER_SK
WHERE
(CASE
   WHEN B.OMD_EFFECTIVE_DATETIME > C.OMD_EFFECTIVE_DATETIME
   THEN B.OMD_EFFECTIVE_DATETIME
   ELSE C.OMD_EFFECTIVE_DATETIME
 END) -- greatest of the two effective dates
 <
(CASE
   WHEN B.OMD_EXPIRY_DATETIME < C.OMD_EXPIRY_DATETIME
   THEN B.OMD_EXPIRY_DATETIME
   ELSE C.OMD_EXPIRY_DATETIME -- smallest of the two expiry dates
 END)
```

This query provides you a full history result showing the raw output for a Customer Hub and its two Satellites. The first Effective Date/Time and Expiry Date/Time combination (column 2 and 3) represent the smallest time slices for this set of tables. The other two combinations (columns 4+5 and 6+7) represent how the two Satellites fit in these timelines and the red circles indicate that for a specific 'most-granular' time slice there are were no changes for that specific Satellite (repeating of values).

	CUSTOMER_SK	OMD_EFFECTIVE_DATETIME	OMD_EXPIRY_DATETIME	OMD_EFFECTIVE_DATETIME	OMD_EXPIRY_DATETIME	OMD_EFFECTIVE_DATETIME	OMD_EXPIRY_DATETIME
1	826BC4A25D1254815FE2FC758204CB18	2016-08-26 19:14:22.4099104	2016-08-26 21:14:22.7418712	2016-08-26 19:14:22.4099104	2016-08-26 21:14:22.7418712	2016-08-26 19:14:22.4099104	2016-08-26 21:14:22.7418712
2	826BC4A25D1254815FE2FC758204CB18	2016-08-26 21:14:22.7418712	2016-08-26 21:15:02.3823868	2016-08-26 21:14:22.7418712	2016-08-26 21:15:02.3823868	2016-08-26 21:14:22.7418712	2016-08-26 21:15:50.4674985
3	826BC4A25D1254815FE2FC758204CB18	2016-08-26 21:15:02.3823868	2016-08-26 21:15:50.4674985	2016-08-26 21:15:02.3823868	2016-08-26 21:16:15.8417474	2016-08-26 21:14:22.7418712	2016-08-26 21:15:50.4674985
4	826BC4A25D1254815FE2FC758204CB18	2016-08-26 21:15:50.4674985	2016-08-26 21:16:15.8417474	2016-08-26 21:15:02.3823868	2016-08-26 21:16:15.8417474	2016-08-26 21:15:50.4674985	2016-08-26 21:16:15.8417474
5	826BC4A25D1254815FE2FC758204CB18	2016-08-26 21:16:15.8417474	9999-12-31 00:00:00.0000000	2016-08-26 21:16:15.8417474	9999-12-31 00:00:00.0000000	2016-08-26 21:16:15.8417474	9999-12-31 00:00:00.0000000
6	AE3E0495C70AE96A519B8D01C72C5585	2016-08-26 19:14:22.4099114	9999-12-31 00:00:00.0000000	2016-08-26 19:14:22.4099114	9999-12-31 00:00:00.0000000	2016-08-26 19:14:22.4099114	9999-12-31 00:00:00.0000000
7	D6C47621D0E5FC4206FD57D18E35A172	2016-08-26 19:14:22.4099144	9999-12-31 00:00:00.0000000	2016-08-26 19:14:22.4099144	9999-12-31 00:00:00.0000000	2016-08-26 19:14:22.4099144	9999-12-31 00:00:00.0000000
8	E7DA4816E8228C0B08AC2102B5A7E900	2016-08-26 19:14:22.4099134	9999-12-31 00:00:00.0000000	2016-08-26 19:14:22.4099134	9999-12-31 00:00:00.0000000	2016-08-26 19:14:22.4099134	9999-12-31 00:00:00.0000000
9	EB67E26BC5B4E89307A2176D30046960	2016-08-26 19:14:22.4099124	9999-12-31 00:00:00.0000000	2016-08-26 19:14:22.4099124	9999-12-31 00:00:00.0000000	2016-08-26 19:14:22.4099124	9999-12-31 00:00:00.0000000

This is the most basic PIT table (albeit at the most granual level – see introduction), which can be persisted and updated as part of the Data Vault refresh mechanisms. The idea is that this table can be used to INNER JOIN Satellites to retrieve any required attributes for upstream processing. The fact that the complex join logic is already handled, and that an INNER JOIN can be used can make a performance difference. Of course, SQL has GREATEST and LEAST functions that you can use as well as opposed to a CASE statement but I thought this was the easiest way to explain.

Making joining easier with multiple tables

In any case, the above technique is geared towards combining two time-variant sources and it is not very straightforward (transparent) to add more time-variant tables using this approach. Of course you can wrap the above logic in a sub-query and merge this with the next time-variant set, but this creates complex logic – especially when handling exceptions such as multi-active Satellites. A better way in my view is to adopt the gaps-and-islands technique again. This basically works as follows:

1) The first step is to create a combined set of all available times (Effective Date/Tiems). This is achieved by UNION-ing the Effective Date/Times across all the time-variant tables (data sets) you need and provides you with the smallest (most granular) available time-slices / level of detail. It is recommended to also UNION a zero-record (ghost record) in this step to make sure a complete timeline between 'earliest history' and 'end of calendar' can be created. This way you don't even need zero-records in the Satellites!
2) The second step is to derive the Expiry Date/Times from the above set, creating a range that can be easily joined against.
3) The third step is to join each individual (time variant) table against this range of time-slices and selecting the attributes you want to present. The join uses the same mechanism as above (greatest of the two effective dates < smallest of the two expiry dates) but the big difference is that each time-variant table is joined against the central 'range' set, making this a lot easier to configure and extend. Additional tables can be easily added (but will have an impact on the granularity). I also recommend to join in the central Hub to the range because typically you want to add the business key itself here or also make mention of the time-slice invoked by the creation of the business key. A business key may have been created some time before the first context started to appear in Satellites.

An example is provided here, and the result is the same as earlier (although I selected some other attributes). You can see how easy it is to add more tables to join using this structure, and it's also easy to add or remove attributes.

```
SELECT
    HUB_CUSTOMER.CUSTOMER_SK AS [HUB_CUSTOMER.CUSTOMER_SK],
    SAT_CUSTOMER.OMD_EFFECTIVE_DATETIME AS [SAT_CUSTOMER.OMD_EFFECTIVE_DATETIME],
    SAT_CUSTOMER.OMD_EXPIRY_DATETIME AS [SAT_CUSTOMER.OMD_EXPIRY_DATETIME],
    SAT_CUSTOMER.SURNAME AS [SAT_CUSTOMER.SURNAME],
    SAT_CUSTOMER_FAST_CHANGING.CONTACT_NUMBER AS [SAT_CUSTOMER_FAST_CHANGING.CONTACT_NUMBER],
    SAT_CUSTOMER_FAST_CHANGING.OMD_EFFECTIVE_DATETIME AS [SAT_CUSTOMER_FAST_CHANGING.OMD_EFFECTIVE_DATETIME],
    SAT_CUSTOMER_FAST_CHANGING.OMD_EXPIRY_DATETIME AS [SAT_CUSTOMER_FAST_CHANGING.OMD_EXPIRY_DATETIME],
    PIT_EFFECTIVE_DATETIME
FROM
(
    SELECT
        CUSTOMER_SK,
        PIT_EFFECTIVE_DATETIME,
        LEAD(PIT_EFFECTIVE_DATETIME,1,'9999-12-31') OVER (PARTITION BY CUSTOMER_SK ORDER BY PIT_EFFECTIVE_DATETIME ASC) AS PIT_EXPIRY_DATETIME
    FROM
    (
        SELECT CUSTOMER_SK, CONVERT(datetime2(7), '1900-01-01') AS PIT_EFFECTIVE_DATETIME FROM  HUB_CUSTOMER
        UNION
        SELECT CUSTOMER_SK, OMD_FIRST_SEEN_DATETIME AS PIT_EFFECTIVE_DATETIME FROM HUB_CUSTOMER
        UNION
        SELECT CUSTOMER_SK, OMD_EFFECTIVE_DATETIME AS PIT_EFFECTIVE_DATETIME FROM SAT_CUSTOMER
        UNION
        SELECT CUSTOMER_SK, OMD_EFFECTIVE_DATETIME AS PIT_EFFECTIVE_DATETIME FROM SAT_CUSTOMER_FAST_CHANGING
    ) PIT
) TimeRanges
INNER JOIN HUB_CUSTOMER
ON TimeRanges.CUSTOMER_SK = HUB_CUSTOMER.CUSTOMER_SK
LEFT OUTER JOIN SAT_CUSTOMER
ON SAT_CUSTOMER.CUSTOMER_SK = TimeRanges.CUSTOMER_SK
AND SAT_CUSTOMER.OMD_EFFECTIVE_DATETIME <= TimeRanges.PIT_EFFECTIVE_DATETIME
AND SAT_CUSTOMER.OMD_EXPIRY_DATETIME >= TimeRanges.PIT_EXPIRY_DATETIME
LEFT OUTER JOIN SAT_CUSTOMER_FAST_CHANGING
ON SAT_CUSTOMER_FAST_CHANGING.CUSTOMER_SK = TimeRanges.CUSTOMER_SK
AND SAT_CUSTOMER_FAST_CHANGING.OMD_EFFECTIVE_DATETIME <= TimeRanges.PIT_EFFECTIVE_DATETIME
AND SAT_CUSTOMER_FAST_CHANGING.OMD_EXPIRY_DATETIME >= TimeRanges.PIT_EXPIRY_DATETIME
```

The attributes you want to show in your PIT or Dimension table

Deriving the expiry date to support easy joining

Creating the overview of available time slices

Joining the various tables back into the time slices

Results | Messages

HUB_CUSTOMER.CUSTOMER_SK	SAT_CUSTOMER.OMD_EFFECTIVE_DATETIME	SAT_CUSTOMER.OMD_EXPIRY_DATETIME	SAT_CUSTOMER.SURNAME	SAT_CUSTOMER_FAST_CHANGING.CONTACT_NUMBER	SAT_CUSTOMER_FAST_CHANGING.OMD_
826BC4A25D1254815FE2FC758204CB18	2016-08-26 19:14:22.4099104	2016-08-26 21:14:22.7418712	Man	99999	2016-08-26 19:14:22.4099104
826BC4A25D1254815FE2FC758204CB18	2016-08-26 21:14:22.7418712	2016-08-26 21:15:02.3823868	Man	99999	2016-08-26 21:14:22.7418712
826BC4A25D1254815FE2FC758204CB18	2016-08-26 21:15:02.3823868	2016-08-26 21:16:15.8417474	Vos	99999	2016-08-26 21:14:22.7418712
826BC4A25D1254815FE2FC758204CB18	2016-08-26 21:15:02.3823868	2016-08-26 21:16:15.8417474	Vos	412345	2016-08-26 21:15:50.4674985
826BC4A25D1254815FE2FC758204CB18	2016-08-26 21:16:15.8417474	9999-12-31 00:00:00.0000000	Vos	9874634	2016-08-26 21:16:15.8417474
AE3E0495C70AE96A519B8D01C72C5585	2016-08-26 19:14:22.4099114	9999-12-31 00:00:00.0000000	Doe	41234	2016-08-26 19:14:22.4099114
D6C47621D0E5FC4206FD57D18E35A172	2016-08-26 19:14:22.4099144	9999-12-31 00:00:00.0000000	Evans	89235	2016-08-26 19:14:22.4099144
E7DA4816E8228C0B08AC2102B5A7E900	2016-08-26 19:14:22.4099134	9999-12-31 00:00:00.0000000	Smith	41234	2016-08-26 19:14:22.4099134
EB67E26BC5B4E89307A2176D30046960	2016-08-26 19:14:22.4099124	9999-12-31 00:00:00.0000000	Slimpy	23555	2016-08-26 19:14:22.4099124

Are we there yet?

It is important to realise that the ranges / time-slices created provide the most detailed level available: that is all the history available for all attributes between all involved tables. However, the attributes you select as output may not necessarily be directly related to some of the change records. Or, to look at this the other way: the attribute that may have lead to a change being tracked in the Satellite may not be exposed in the SELECT statement.

Depending on which attributes you expose this may lead to duplicates in the end result, but these can easily be handled by adding row condensing similar to the one used in the Satellites. By applying this concept over the top of the above query it is easy to show the historical context for the attributes in scope and very easy to toggle between a Dimension view, a true Point-in-Time (snapshot) or raw changes.

Applying row condensing for PIT or Dimensions can be implemented by creating a checksum across the all attributes except the lowest-granularity (range) Effective Date/Time, and comparing the rows with each other when sorted by Effective Date/Time. If there are no changes between the rows then the row with the higher Effective Date can be discarded (condensed). This is because apparently the change in the Satellite that created the time-slice was not required in this specific output. The query logic is a bit long to paste here, but can be tested in the Virtual EDW app.

The key steps are to:

- From the output of the time-variant query shown in the previous section; create a checksum (I use MD5) across all all attributes excluding the Effective Date/Time for the range.
- Make the checksums available for comparison, for example:

  LAG(ATTRIBUTE_CHECKSUM, 1, '-1') OVER(PARTITION BY [HUB_CUSTOMER.CUSTOMER_SK] ORDER BY PIT_EFFECTIVE_DATETIME ASC) AS PREVIOUS_ATTRIBUTE_CHECKSUM

- Filter out records were the checksum is different than the previous checksum (these are the duplicates)

Again, this applies to every scenario where you only select a subset of all available attributes in time-variant tables.

Final thoughts

The above explanation should provide some handles to create your own time-variant output, but there are always other things to be aware of when deploying these solutions. This is especially true in cases where the result is instantiated into a physical table that needs to be periodically refreshed. Without going into too much detail (posts get too long) here are some pointers:

- Updates in Satellites may occur at different times, which can mean that a later Effective Date/Time can be available while another Satellite still needs to be updated. I work in an environment where everything runs continuously, which means that the moment of refresh for a PIT or Dimension may mean some information may still be loaded into some of the Satellites. This is obviously only an issue if you want to process deltas into your PIT or Dimension, but it is good practice to build in delta handling from the start as a full refresh may quickly become too costly. There are a few ways to handle this (good topic for a next post). The solution can be to implement a mechanism that rolls-back some of the rows and re-applies the updated history or to create loading dependencies (not my personal favourite).
- In complex environments the queries may not work in a single pass, but it's easy to break up the steps into separate ETL processes that each persist their data. This can give the database a bit of breathing space if required.

As I mentioned earliest in this post I prefer to 'jump' from a Data Vault / DWH into a Dimension or other form of Presentation Layer output straight away skipping PIT tables altogether.