Getting the data out again

An often-mentioned comment on Data Vault is that while it is relatively easy to load data *into* the solution, it can be hard to get the data out again. This is true in a way, but not specifically related to Data Vault as a concept or methodology.

Managing data is a complex matter regardless of the adopted design approach, and the complexities of designing a Data Warehouse solution (ecosystem) will always need to be addressed somewhere.

One of the distinctions between various Data Warehouse methodologies, is *where* certain complexities are addressed. Or in other words, where (and how) certain Data Warehousing concepts are implemented in the overall architecture. Decisions related to this overall design can cover topics such as where to handle key distribution, when business logic is to be applied, how exceptions can be handled, how data can be structured and presented for delivery and how historised and late-arriving data are going to be managed.

These concepts interact with each other relative to their implementation as part of the architecture. For example, by implementing business logic early on you may need to deal with data that does not comply with the implemented logic upfront. This may require the implementation of exception handling mechanisms which in turn have an impact on the timeliness of data availability, as rejected data needs to be delayed in being available to consumers.

Another example is the decision to select a technical or a functional (business) timeline to organise data against, and of course which one.

From a technical perspective this comes down to determining which date / time attribute is used for 'effective dating' and usually making this part of the Primary Key of the historised (time-variant) table.

Functional timelines make more sense to data end consumers, and simplify delivery when these are standardised earlier in the solution.

However, standardising on a functional timeline early on in the architecture, and using this to historise data means that you become subject to potential data quality and late-arriving data issues.

A Data Warehouse solution relies on data being delivered by other systems, and the content and quality of the provided data is rarely something the Data Warehouse team can control. Conversely, a technical Data Warehouse timeline is more robust because the values *can* be controlled by the Data Warehouse environment – but by itself a technical timeline has limited usefulness for data consumers. Data usually needs to be interpreted before made available.

Decisions like these about the organisation of these various concepts into an overall architecture, usually rooted in one of the existing methodologies, make that the overall design needs to be a careful selection of options with explicit consideration of the various pros and cons. These decisions are based on balancing various beliefs and viewpoints around ease of use, performance, ability to refactor, speed of delivery, ability to scale and balancing more generally simplicity versus complexity.

Data Vault, and many similar 'hybrid' approaches, focus on a solution design that has a highly structured and resilient 'back-room' to ingest data and a flexible 'front-room' where business logic and requirements can be iteratively clarified in conjunction with business subject matter experts. The back-room is designed to be as unbreakable as possible, whilst the front-room deals with applying context and interpretation of data. A separation of concerns.

This marks a departure from earlier paradigms on Data Warehousing that prescribed (albeit implicitly) that data should be judged (interpreted) on the way 'in' to the solution. A Data Vault based solution interprets data 'on the way out', which is another way of saying that business logic (the implementation of business requirements) is applied when delivering Data Marts. The selection of the (functional) timeline to deliver data against is one of these interpretations, with far-reaching consequences.

This design decision means that the processes that deliver Data Marts have to be able to correctly merge various data sets that contain time-variant (historised) information, as well as applying various levels of interpretation including the correct representation of changes across the selected timeline.

This paper explains a pattern to deliver Data Marts on top of a Data Vault model in a flexible way, and focuses on a core element to arrive at a suitable business outcome: handling the interpretation of time from a business perspective.

At its core, this paper explains why the Load Date / Time Stamp (LDTS) is usually *not* a suitable timeline for reporting. In most cases it should *only* be used for filtering purposes, where it can be used for great effect to create a true 'Data Warehouse time machine', and in support of highly automated exception handling of data processing household tasks. A technical design of the data loading logistics that can even gracefully handle late-arriving data.

Ongoing clarification of requirements

A Data Mart of any kind, in the Data Vault context at least, can be considered technically correct if it is loaded in a deterministic way. This means that if you truncate the table and reload it entirely, it should contain the exact same information. The ability to reload a table this way is considered a requirement for agility.

One of the ideas behind Data Vault is the ability to quickly deploy new Data Marts, or at least new versions of existing ones, to keep up with the speed of the business. In simpler terms, the ability to refactor the data in a new or updated Data Mart contributes greatly to reduce the time between when the business subject matter experts have clarified requirements and when they can see and validate the outcomes from their input.

This responsiveness is a core technical function to achieve scalable data logistics when the number of source interfaces grows. It is enabled by separating the storage of time-stamped raw data (organised in its correct business context) in a Data Vault with the application of business logic isolated in Data Marts. By retaining the original values (i.e. prior to the logic being applied) it becomes possible to 'reroll' a Data Mart when business logic is applied or updated.

In Data Vault, the separation of the back-room and front-room means that three things need to happen for a traditional Data Mart delivery. First of all, data (attributes) from various historised data sets (i.e. Satellites) needs to be combined into a single unified historised data set. Secondly, data needs to be reorganised against the desired timeline for delivery. And thirdly, transformations need to be applied based on the data set that is being processed.

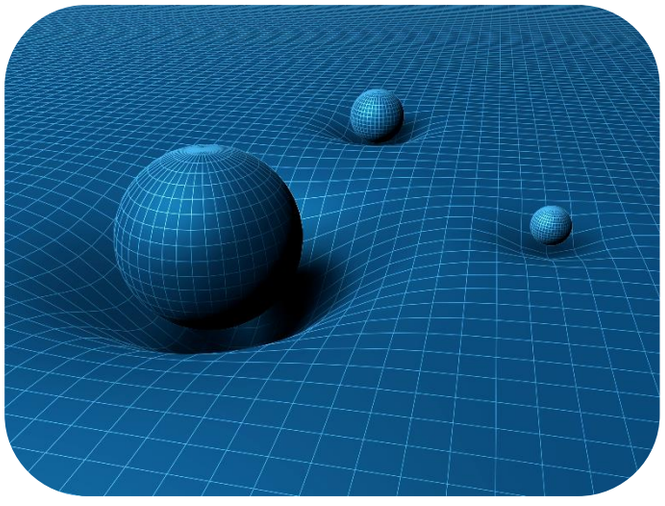
The first two activities go hand-in-hand. To create a Data Mart, we need to use the technical back-room date / time information for filtering and selection and decide on a functional (business) timeline for reporting.

I want to clarify that creating a Dimension does not necessarily require all data (attributes) across Satellites to be merged to deliver a single historised data set (i.e. a Dimension). You pick and choose the attributes you need to deliver to the requirements and build you Dimensions (and Fact tables) accordingly. In other words, just because a Hub as a few Satellites does not dictate that they will be delivered as a single Dimension output.

In Data Vault, Satellites may be split for various reasons (i.e. differences between systems, security, arrival etc.). It is even possible that attributes that describe a specific Business Concept (Hub) will be added to a different Dimension altogether.

What time is it?

There are essentially two types of 'time' in Data Vault: the Load Date / Time Stamp (LDTS) and everything else. The LDTS is the moment that any data delta record or transaction has arrived in the Data Warehouse environment. In a more traditional Data Vault environment the LDTS is set when the data enters the Data Vault model but you can also choose to capture the LDST when the data enters the Persistent Staging Area (PSA). The benefit of using the latter is that it allows you to refactor your entire Data Warehouse (not just the Data Marts) following the Virtual Data Warehouse concept.



Either way, this is usually implemented as a default computed value (i.e. 'now') which is set by the RDBMS when a row is written to the table - assuming you have an RDBMS of course. The LDTS is part of the Primary Key of the historised tables in the Data Warehouse environment such as PSA tables and Satellites. This implies that for data in the 'back-room', the data is organised along the perspective of the *arrival* of data into the Data Warehouse.

Any other date and time information will be treated as context, descriptive information, and stored as regular attributes that describe the (Business) key and are subject to being tracked for changes over time. The 'time' being the LDTS, in this case.

Data that arrives later will get a later (higher) LDTS, and therefore will be treated as 'later in time' along with any other attributes that arrived with the data set.

This is how time-variant data is managed in the back-room of the solution. This approach provides some valuable auditing capabilities, but for most business users and use-cases, reporting against the timeline of 'arrival of information' is not very helpful.

Indeed, in most cases the requirement is to provide 'business' timelines that shows users the information they seek. Dates and times that correspond with business processes and system information that is unrelated to Change Data Capture or data integration times. For example, it is entirely possible for data to get delayed on its way into the Data Warehouse. In some cases, one transaction can even 'overtake' another in transit to the Data Warehouse and subsequently get a later LDTS, even though the event took place earlier in the real world.

In other cases, changes committed in the system landscape 'now' may apply to data at earlier (or later) points in time – an activity known as backdating or future-dating. A user can adjust details in the present, and these changes may be loaded into the Data Warehouse with only a few milliseconds delay (and thus with a LDTS that is only marginally higher than the committed change) but doing so can impact data that is attributed to a business period years ago. This happens frequently, for instance, when application administrators receive information that prompts them to adjust the (historical) view that users see when opening screens in the application.

This is a key concept: business users may not be aware of this behaviour, as they usually only interact with the application. But the Data Warehouse captures everything, and needs to be able to make sense of this. As a data solution it should be able to provide both the view business users are familiar with, the one they can see in their applications, as well as the true order of changes made for auditability and quality control (and arguably business education).

A key element to get right in delivering Data Marts is to make sure you select the right timeline for your reporting, and using the LDTS for reporting is rarely the desired business outcome.

A case of late arriving data

Data Vault uses the LDTS as the timeline against which changes over time are organised. The LDTS is the effective date in Satellites, and is a part of the Primary Key. But any given Data Warehouse solution may contain many alternative date / time attributes that can be used as timelines for delivery purposes – options for displaying changes over time. These attributes are available as context of a Business Key in Satellites. For example, the 'business' effective date used in an application, or issue dates of documents.

Selecting any timeline for reporting *other* than the LDTS (i.e. in the 'front-room' of the solution) exposes the solution to a perennial challenge in Data Warehousing: late-arriving data.

Late arriving data? A bit tongue-in-cheek you could argue that in a Data Vault solution, there *can* be no late-arriving data. The concept simply doesn't exist. If you consider the strict separation between the 'back-room' and 'front-room' of the Data Warehouse eco system this is correct, but only for the back-room part of the solution where the LDST is the effective date. If you make sure the LDTS is managed by the Data Warehouse environment, and every row is 'timestamped' when it arrives, the LDST is inherently incremental. From this point of view Satellite information will *always* arrive in order and be stored accordingly.

Of course, this does not mean that Data Vault can't 'handle' late-arriving data. In fact, the opposite is true. It just means that the definition, the interpretation, of 'late-arriving data' is from a business timeline perspective. Data Vault provides an elegant way to manage late arriving data this way. The order of arrival is captured sequentially (using the LDTS), but the point in time to which the data applies can be back-dated or future-dated.

This is one of the main reasons why the back-room is so robust: the LDTS as effective date in the back-room is managed by the Data Warehouse environment itself and therefore can by definition only ever increase in value. Because it is fully under the control of the Data Warehouse it cannot be affected by shifts in time zones or tampering by source system administrators or business users.

If business users' requirements dictate that data needs to be reported against business timelines, it implies that the LDTS cannot be used as this only shows the order of *arrival* of data. Changes over time would therefore have to be recalculated (reordered) against the selected business timeline.

And as mentioned earlier, any timeline which is not the LDTS (or similar technical date / times controlled by the Data Warehouse environment such as ETL processing times) means that provisions need to be in place to manage late-arriving data as the contents of this data are outside the control of the Data Warehouse – you are subject to whatever the source systems provide.

To summarise: late-arriving data in this context is from the perspective of the *selected timeline*. Even though data arrives in incremental order in the Data Vault, the data may pertain to a different time-period. This will come into effect when changes over time are re-arranged according to this new timeline.

This is also one of the main reasons why end-dating records (setting the expiry date / time) in the Data Vault is not recommended. You would need to recalculate timelines for the selected business timelines anyway. And even if you *do* report against the LDST, chances are that due to row condensing you still find that a recalculation of the expiry date / time is required, because after the condensing has been applied the timelines do not add up anymore.

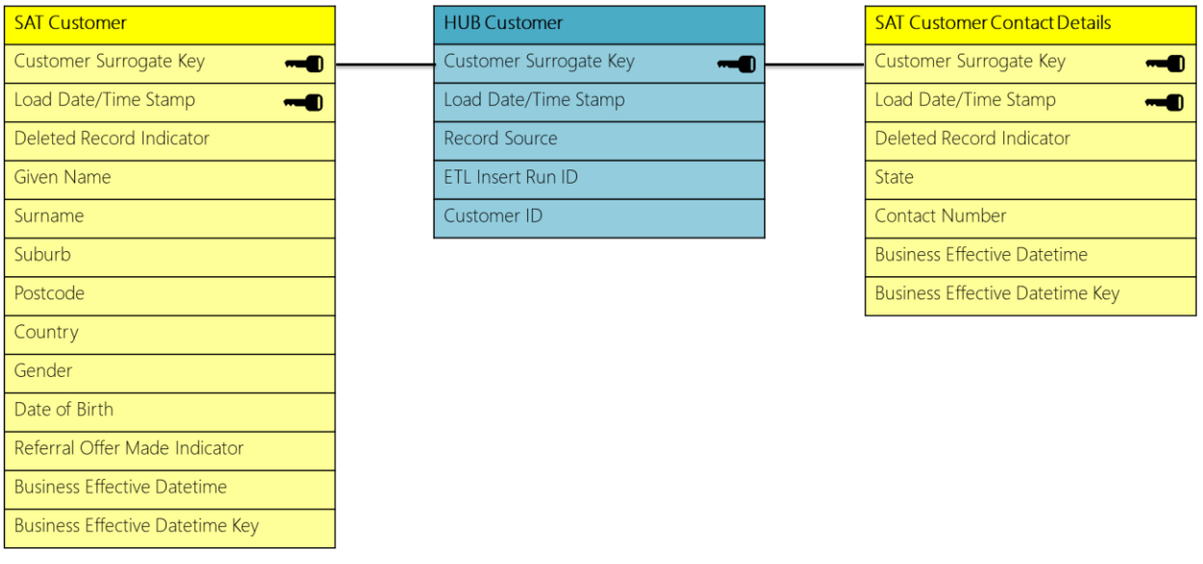
There are more reasons why end-dating records in the Data Vault is not recommended. For instance, there is a performance cost for running physical UPDATE statements over large data sets, and it does not scale well. Also, expiry dates are often misinterpreted and used to deliver information. Remember, the expiry date for a LDST effective date is still the LDST timeline!

An 'Insert-Only' Data Vault really seems to be the best, or maybe even only way to go, or else you would risk losing the coveted robustness of data ingestion.

An example of late-arriving data, from a business perspective

Consider the following example.

We have modelled the Data Vault to separate the main customer details from the contact details into two separate Satellites. Both Satellites provide context to the Customer Business Key (Customer ID), as embodied in the Hub Customer table. For ease of presentation I have removed some of the standard ETL control framework attributes such as the Full Row Hash, the Source Row ID, the Record Source and the pointer to the ETL process instance that inserts the data.



The following data is loaded. On…

- 2018-01-01 the Business Key first appears in the Hub, but there is no context available yet. This could happen because the Business Key was used in another area, such as a sales transaction.
- 2018-09-20 at 09:00 the first context is made available and loaded into the Sat Customer. This identifies the customer, we have a name and some information. This is effective-dated in the operational system as per 2017-11-20 08:15, which means that in the front-end application of the operational system users will see the information as per this date.
- 2018-09-20 at 09:15 contact details are made available, effective-dated in the operational system as per 2018-07-01.
- 2018-11-20 at 11:26 a change to contact details comes through. However, this is backdated to 2016-04-01 which indicates that the latest received contact details were supposed to have been received at this date.

These events are represented in 'data' in the table below, meant as a simple overview to explain the use case. The complete sample data including table structure, contents as well as the logic to deliver this into a Dimension can be downloaded here.

Load Date / Time Stamp (LDTS)	Business Effective Date / Time	Event
2018-01-01 00:00:00.0000000	-	The Business Key is created in the Hub, with Customer ID CUST_4. It's all we know about this customer at this stage.
2018-09-20 09:00:00.0000000	2017-11-20 08:15:00.0000000	The first row is created in Sat Customer. We now know that CUST_4 is 'Jonathan Slimpy' and have various other details such as Gender and Date of Birth. This is effective as of 2017-11-20 08:15 in the operational systems, although it was received at 2018-09-20 09:00.
2018-11-20 09:15:00.0000000	2018-07-01 00:00:00.0000000	The first row in the Sat Customer Contact Details is created with the additional details that were made available. Contact Number information (value 23555) has been received at 2018-11-20 09:15, but is effective-dated in the systems as per 2018-07-01.
2018-11-20 11:26:00.0000000	2016-04-01 00:00:00.0000000	A second row of additional details is presented for Sat Customer Contact Details. The contact number changes to 23510, but applies to an earlier period (2016-04-01).

As you can see, the information arrives in the Data Vault (and is stored in) the order of the LDTS. However, a user would probably want to see the changes as per the business effective date / time. The pattern to combine this data, these three tables in this example, into a Dimension needs to be able to merge the changes (over time) of these tables and reorder the changes across the selected business effective date timeline.

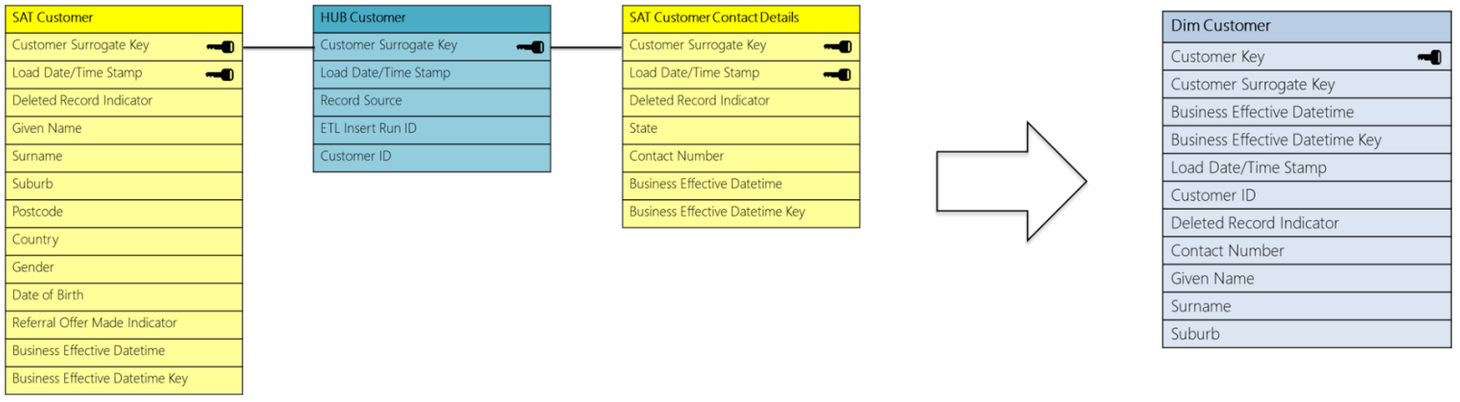
As additional requirements the pattern needs to provide for:

- A performant selection. Combining potentially large (time-variant) data sets can be a resource expensive operation.
- The ability to virtualise as well as load incrementally. If we would load the entire set all at once, the result should be the exact same as if the table would have been populated incrementally.

You may notice both a Business Effective Date / Time attribute as well as a Business Effective Date / Time Key. The 'Key' attribute is a technical version of the Business Effective Date / Time to make sure the values are unique. Since we are switching timelines, by virtue of reordering data from the LDTS timeline to the Business timeline, we no longer have full control of the values that we're ordering our data against, as these are provided from our feeding systems.

This means we may encounter issues around multiple records being 'active' at a given point-in-time when using the Business timeline, caused by overlapping or non-unique dates and times. The 'Key' concept addresses this issue by applying 'uniquefication' on the selected timeline by adding details that are unique by definition, such as the Source Row ID. To keep things as simple as possible I will use the Business Effective Data / Time for the purposes of explaining the concept.

The resulting Dimension table combines the information from the three tables and looks like this:



This Customer Dimension combines the Business Key (from the Hub) with the Contact Number (from the Sat Customer Contact Details) and the Given Name, Surname and Suburb (from the Sat Customer). The evaluation of the Deleted Record Indicator is done based on information in the Sat Customer, but this is out of scope for this paper. This is in itself a relatively complex topic. Suffice it to say here that the Deleted Record Indicator provided by the source systems, is treated as any regular attribute that can change over time and only applies to the scope of information in the corresponding Satellite.

For demonstration purposes, the Customer Key is just a simple sequence number. The Customer Surrogate Key and Customer ID are respectively the hashed and natural Business Keys retrieved from the Hub.

A pattern for delivering a Data Mart

If we would load incrementally in four separate executions the result would look like the below (the changes are highlighted, and some attribute names have been abbreviated).

Run 1 (selecting available data from the dawn of time up to and including 2018-09-20 09:00):

CUSTOMER_SK	BUSS_EFF_DT	BUSS_EFF_DT_KEY	LDTS	CUST_ID	DEL_REC_IND	CONT_NUMBER	GIVEN_NAME	SURNAME	SUBURB
0xB31A~	1900-01-01 00:00	19000101000000000000000000	1900-01-01 00:00	CUST_4	NULL	NULL	NULL	NULL	NULL
0xB31A~	2017-11-20 08:15	20171120081500000000000000	2018-09-20 09:00	CUST_4	N	NULL	**Jonathan**	**Slimpy**	**London**

Run 2 (selection available data from the 2018-09-20 09:00 up to and including 2018-11-20 09:15):

CUSTOMER_SK	BUSS_EFF_DT	BUSS_EFF_DT_KEY	LDTS	CUST_ID	DEL_REC_IND	CONT_NUMBER	GIVEN_NAME	SURNAME	SUBURB
0xB31A~	1900-01-01 00:00	19000101000000000000000000	1900-01-01 00:00	CUST_4	NULL	NULL	NULL	NULL	NULL
0xB31A~	2017-11-20 08:15	20171120081500000000000000	2018-09-20 09:00	CUST_4	N	NULL	Jonathan	Slimpy	London
0xB31A~	2018-07-01 00:00	20180701000000000000000000	2018-11-20 09:15	CUST_4	N	**23555**	Jonathan	Slimpy	London

Run 3 (selection available data from the 2018-11-20 09:15 up to and including 2018-11-20 11:26 – or 9999-12-31 for that matter):

CUSTOMER_SK	BUSS_EFF_DT	BUSS_EFF_DT_KEY	LDTS	CUST_ID	DEL_REC_IND	CONT_NUMBER	GIVEN_NAME	SURNAME	SUBURB
0xB31A~	1900-01-01 00:00	19000101000000000000000000	1900-01-01 00:00	CUST_4	NULL	NULL	NULL	NULL	NULL
0xB31A~	2016-04-01 00:00	20160401000000000000000000	2018-11-20 11:26	CUST_4	NULL	23510	NULL	NULL	NULL
0xB31A~	2017-11-20 08:15	20171120081500000000000000	2018-09-20 09:00	CUST_4	N	NULL	Jonathan	Slimpy	London
0xB31A~	2018-07-01 00:00	20180701000000000000000000	2018-11-20 09:15	CUST_4	N	23555	Jonathan	Slimpy	London

Without adding any further interpretation and post-processing, the result takes into account the late-arriving data as visible in the 2nd row. This behaviour is intended - it is exactly what has happened over time. Of course, it is possible to massage this information further by filling in any unwanted NULL values, but these are business rules that require further clarification – arguably this would 'create' information that may not have been available originally.

Delivering this data set in a way that is both performant and 'virtualisation ready' can be achieved by allowing date range selection which uses the LDTS as a filtering mechanism. A good approach to implement this technically is to use a Table Valued Function. This kind of function works very much the same as a view but accepts parameters (the from- and to date / time in this case) and returns the results as a table. You can query this the same as any other object in the database, as long as you provide the required parameters.

The following query would return the available delta between the start of time and the LDTS of 2018-09-20 09:00.

```
SELECT *
FROM DIM_CUSTOMER_FN('1899-01-01T00:00:00', '2018-09-20 09:00:00')
WHERE CUSTOMER_SK = 0xB31AA746C00064CEC4D0281ED62FB432
```

The result would be:

CUSTOMER_SK	BUSS_EFF_DT	BUSS_EFF_DT_KEY	LDTS	CUST_ID	DEL_REC_IND	CONT_NUMBER	GIVEN_NAME	SURNAME	SUBURB
0xB31A~	1900-01-01 00:00	19000101000000000000000000	1900-01-01 00:00	CUST_4	NULL	NULL	NULL	NULL	NULL
0xB31A~	2017-11-20 08:15	20171120081500000000000000	2018-09-20 09:00	CUST_4	N	NULL	Jonathan	Slimpy	London

Similarly, running the following query only shows the latest change:

```
SELECT *
FROM DIM_CUSTOMER_FN('2018-11-20 09:15:00', '9999-01-01T00:00:00')
WHERE CUSTOMER_SK = 0xB31AA746C00064CEC4D0281ED62FB432
```

This time, the result is::

CUSTOMER_SK	BUSS_EFF_DT	BUSS_EFF_DT_KEY	LDTS	CUST_ID	DEL_REC_IND	CONT_NUMBER	GIVEN_NAME	SURNAME	SUBURB
0xB31A~	2016-04-01 00:00	20160401000000000000000000	2018-11-20 11:26	CUST_4	NULL	23510	NULL	NULL	NULL

And finally, if we would virtualise the entire Dimension we would simply select the complete time range according to the LDTS (i.e. no filtering applied on LDTS):

```
SELECT *
FROM DIM_CUSTOMER_FN('1899-01-01T00:00:00', '9999-01-01T00:00:00')
WHERE CUSTOMER_SK = 0xB31AA746C00064CEC4D0281ED62FB432
ORDER BY BUSINESS_EFFECTIVE_DATETIME_KEY
```

This will return the complete Dimension in the same way as if it would have been loaded incrementally, including the late-arriving information.

CUSTOMER_SK	BUSS_EFF_DT	BUSS_EFF_DT_KEY	LDTS	CUST_ID	DEL_REC_IND	CONT_NUMBER	GIVEN_NAME	SURNAME	SUBURB
0xB31A~	1900-01-01 00:00	19000101000000000000000000	1900-01-01 00:00	CUST_4	NULL	NULL	NULL	NULL	NULL
0xB31A~	2016-04-01 00:00	20160401000000000000000000	2018-11-20 11:26	CUST_4	NULL	**23510**	NULL	NULL	NULL
0xB31A~	2017-11-20 08:15	20171120081500000000000000	2018-09-20 09:00	CUST_4	N	NULL	**Jonathan**	**Slimpy**	**London**
0xB31A~	2018-07-01 00:00	20180701000000000000000000	2018-11-20 09:15	CUST_4	N	**23555**	Jonathan	Slimpy	London

How does this work? By using the LDTS *only as filtering mechanism*, unless you are actually reporting on the arrival of information for auditing purposes. In other words, the parameters provided to the Dimension logic are always LDTS values.

First off, the range of changes is selected. This is where the initial filtering is applied as you will see in the sample query further down. To do this, select all the changes (unique records) across all involved tables that have a LDTS between the provided parameters. This is a union of all changes of all tables, so the union of in-scope rows of the Sat Customer table with the in-scope rows of the Sat Customer Additional Details in this example.

This union provides you the subset of all changes where the LDTS is greater than the input variable (the start of the load window) and the LDTS is smaller or equal than the variable that represents the top of the load window. Additional filtering can be applied here as well - including further row condensing across time if required, similar to what would happen in the Satellite loading pattern. This is conceptually similar to how Satellite and Persistent Staging Area patterns work and not explained any further here.

This will provide the list of records that contain a (true) change within the designated window.

The second step is to join the required tables back to the range data set from the first step in order to add the attributes. However, now the data is joined against the *business* timeline because this is the timeline that the data needs to be represented against for information consumption purposes.

To achieve this, apply the following (join) conditions:

- Join the table you want to add on the Business Key, where,
- The range Business Date / Time is greater or equal than the Business Effective Date / Time in the joining table (i.e. a Satellite).
- The range Business Date / Time is smaller than the Business Expiry Date / Time in the Satellite.
- The last join condition is that the range LDTS is greater or equal than the LDTS in the joining table. This final condition makes sure the filtering is applied on the joining tables as well for performance.

The Business Expiry Date / Time is recommended to be calculated (derived) on the fly when joining the table. In most cases this is a better trade-off in performance compared to storing the expiry dates on disk. Besides, it is not always clear which date is required – and you may need to report against different timelines for different Data Marts, which would mean you would have to store multiple expiry dates!

Lastly, don't forget about potential row condensing requirements which will reduce the rows and therefore require a re-calculation of Expiry Date / Times.

An example query is provided below, but please have a look at the complete query which can be downloaded here including the scripts to create the sample data. I had to reduce the query to be able to fit it in the paper.

```
CREATE FUNCTION DIM_CUSTOMER_FN(@start_datetime DATETIME2, @end_datetime DATETIME2) RETURNS TABLE AS RETURN

WITH All_Changes AS (
  SELECT -- Sat Customer
      [SAT_CUSTOMER].[CUSTOMER_SK]
    , [SAT_CUSTOMER].[BUSINESS_EFFECTIVE_DATETIME]
    , [SAT_CUSTOMER].[BUSINESS_EFFECTIVE_DATETIME_KEY]
    , [SAT_CUSTOMER].[LOAD_DATETIME]
  FROM SAT_CUSTOMER
  WHERE LOAD_DATETIME] > @start_datetime AND LOAD_DATETIME] <= @end_datetime
  UNION
  SELECT -- Sat Customer Details
     [SAT_CUSTOMER_CONTACT_DETAILS].[CUSTOMER_SK]
   , [SAT_CUSTOMER_CONTACT_DETAILS].[BUSINESS_EFFECTIVE_DATETIME]
   , [SAT_CUSTOMER_CONTACT_DETAILS].[BUSINESS_EFFECTIVE_DATETIME_KEY]
   , [SAT_CUSTOMER_CONTACT_DETAILS].[LOAD_DATETIME]
  FROM [SAT_CUSTOMER_CONTACT_DETAILS]
  WHERE LOAD_DATETIME] > @start_datetime AND LOAD_DATETIME] <= @end_datetime
  UNION
  SELECT -- Hub Customer, for zero-record purposes
     [CUSTOMER_SK]
   , CONVERT(DATETIME2(7), '1900-01-01') AS [BUSINESS_EFFECTIVE_DATETIME]
   , CONVERT(NUMERIC(38),  '19000101000000000000000') AS [BUSINESS_EFFECTIVE_DATETIME_KEY]
   , CONVERT(DATETIME2(7), '1900-01-01') AS [LOAD_DATETIME]
   FROM [HUB_CUSTOMER]
   WHERE CONVERT(DATETIME2(7), '1900-01-01') > @start_datetime AND CONVERT(DATETIME2(7), '1900-01-01') <= @end_datetime
), DisplayResults -- Return the net results
AS (
SELECT
    All_Changes.[CUSTOMER_SK]
  , All_Changes.[BUSINESS_EFFECTIVE_DATETIME]
  , All_Changes.[BUSINESS_EFFECTIVE_DATETIME_KEY]
  , All_Changes.[LOAD_DATETIME]
  , [HUB_CUSTOMER].[CUSTOMER_ID]
  , [SAT_CUSTOMER].[DELETED_RECORD_INDICATOR]
  , [CONTACT_NUMBER]
  , [GIVEN_NAME]
  , [SURNAME]
  , [SUBURB]
FROM All_Changes
INNER JOIN [HUB_CUSTOMER] ON All_Changes.[CUSTOMER_SK] = [HUB_CUSTOMER].[CUSTOMER_SK]
LEFT JOIN
(
SELECT *, LEAD([BUSINESS_EFFECTIVE_DATETIME_KEY],1,'99999999999999999999999') OVER (
        PARTITION BY [CUSTOMER_SK] ORDER BY [BUSINESS_EFFECTIVE_DATETIME_KEY] ASC) AS [BUSINESS_EXPIRY_DATETIME_KEY]
FROM [SAT_CUSTOMER]
) SAT_CUSTOMER
    ON [SAT_CUSTOMER].[CUSTOMER_SK] = All_Changes.[CUSTOMER_SK]
   AND All_Changes.[BUSINESS_EFFECTIVE_DATETIME_KEY] >= [SAT_CUSTOMER].[BUSINESS_EFFECTIVE_DATETIME_KEY]
   AND All_Changes.[BUSINESS_EFFECTIVE_DATETIME_KEY] < [SAT_CUSTOMER].[BUSINESS_EXPIRY_DATETIME_KEY]
   AND All_Changes.[LOAD_DATETIME] >= [SAT_CUSTOMER].[LOAD_DATETIME]
LEFT JOIN
(
    <same for Sat Customer Contact Details and any other additional Satellites>
)
/*  <-- ADD FURTHER BUSINESS LOGIC BELOW HERE -->  */
SELECT
    [CUSTOMER_SK]
  , [BUSINESS_EFFECTIVE_DATETIME]
  , [BUSINESS_EFFECTIVE_DATETIME_KEY]
  , [LOAD_DATETIME]
  , [CUSTOMER_ID]
  , [DELETED_RECORD_INDICATOR]
  , [CONTACT_NUMBER]
  , [GIVEN_NAME]
  , [SURNAME]
  , [SUBURB]
FROM DisplayResults
```

When it comes to technology and, more broadly, solution architecture, other techniques may be a better fit to deliver the desired outcome. Table Valued Functions may not be a feasible solution for ultra-large data sets, in which case it may be possible to merge these concepts with PIT and Bridge tables – which are mentioned later in this paper. There are various ways to achieve this outcome, and some may work better for you than others. The same applies to UNION statements. Please consider these as explanations of how the pattern can be implemented, and there are more ways to achieve the same outcome – in some cases in a better way depending on the technical architecture.

So far, we only used Satellites directly linked to a Hub to create a Dimension. Thankfully, extending this out across the Data Vault model is easy. This is necessary if you want to add attributes stored in Satellites that are related to another Hub, or in a Link-Satellite to your Dimension. Adding information from Satellites not directly related to the 'base' Hub requires various joins across Link tables.

Think of these joins as a single data set to join to the range of changes. All of this data can be merged into the Dimension using this pattern by simply incorporating the joins / pathways you use to query everything together as individual time-variant sets the same way as you would add a Satellite. Every data set including joins is in a way a subquery that is added using the same mechanism as the Satellites have been added in the example – including the filtering.

Applying the pattern for Fact Tables, Type 1 and Type 2 Dimensions

The output of the pattern as described in the previous section is fairly generic. It is a consistent way of presenting information to the Presentation Layer, the Data Mart areas. You may have a requirement to present information in a specific way following traditional Kimball classifications of information delivery, or merge the output into an existing Dimension in a specific way.

As you most likely will have seen, the documented pattern already provides the result set as a full history of changes, a Type 2 data set. As such, it can be directly inserted into an existing Type 2 Dimension. Of course, when late-arriving data shows up and you have persisted the (business) expiry dates, these would have to be recalculated. Nothing new here! The filtering using LDST will make sure only new information is appended.

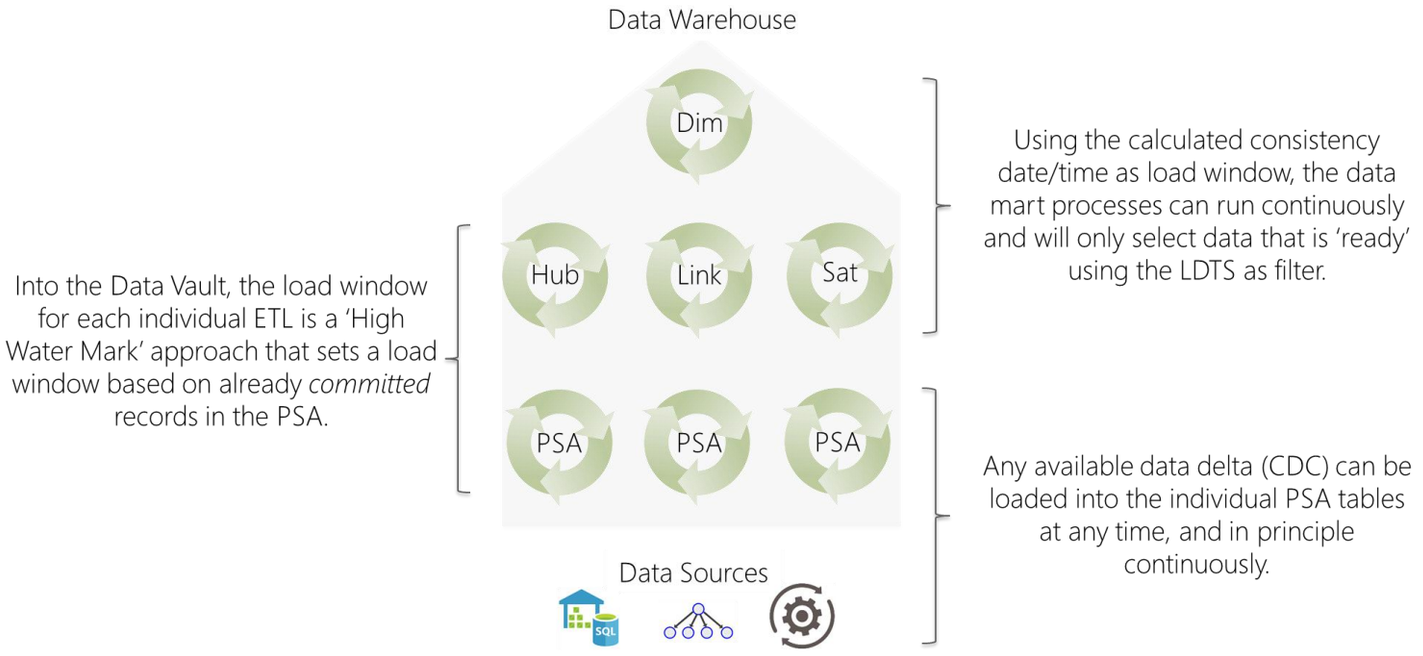
Loading the resulting data set from this pattern into a Type 1 (current state) Dimension is a bit more interesting. If we stick to the concept that we want to report using the business timeline, the merge of the resulting data set (delta from the Data Vault) will require a *conditional update* in order to be applied to the target. The ETL that merges the data should check if the most recent Business Effective Date / Time is equal or greater than the date / time already available in the Type 1 Dimension. Also, the values (full row checksum, or hash) should be different. If these conditions are met, the record can be updated. Otherwise (i.e. if the most recent Business Effective Date / Time is lower than the date / time in the Dimension) the records can be discarded.

It goes without saying that both scenarios are simpler in the Virtual Data Warehouse, or at least when you virtualise the Data Marts. Persisting information always comes with a maintenance overhead.

Using eventual consistency to drive your load windows

Applying the LDTS as a filtering mechanism to populate your Data Marts allows you to use your Data Warehouse as a 'time machine' where you can go back to any point in the past and show exactly what data was available and how it was presented at that moment in history.

Moreover, in combination with the eventual consistency concepts it allows the Data Warehouse solution to 'push through' or 'pull' data as soon as it becomes available – as soon as the data satisfies the integrity criteria. Combining eventual consistency with a 'pull' pattern means that every ETL process in the Data Warehouse architecture is completely independent and only loads data that should be loaded. Consider the following diagram on this topic, where every ETL process is entirely stand-alone and running continuously. Note that for this purpose I have added the PSA to the standard Data Vault architecture.



This is an incredibly flexible and powerful way to orchestrate and automate the loading of data in a Data Warehouse solution, and to refresh the Data Marts with information that is ready for consumption. In my view, this is as good as micro-batching can become.

What about Point-in-Time (and Bridge) tables?

If you are working with Data Vault, you will probably be familiar with Point-in-Time (PIT) and Bridge tables. Without going into the exact inner workings of these in this paper (Dan Linstedt has some really good resources on these geared towards large scale solutions), I should clarify that these are primarily constructs to enhance query performance.

The range selection in the pattern is very similar to a PIT table, and it is eminently doable to replace this component with a physical underlying PIT table if required. In this case, the implementation of filtering can be pushed up into the ETL that loads into the PIT table.

At the end of the day, every time you persist information it comes with an overhead to keep the data up-to-date.

As a performance enhancer by virtue of persisting data, there is no need for PIT objects in the Virtual Data Warehouse (please have a look at additional information on this concept here and here). This is one of the reasons why personally I avoid these, unless there really is no other option. I would like to think the filtering as explained in this paper goes a long way in remediating performance issues.

The Virtual Data Warehouse prototyping software (VEDW) has some example functionality to generate the PIT constructs which shows that the more attributes you add, the more the result begins to look like a Dimension table. Conversely, the more attributes you remove, the more the result resembles a PIT table. It's all fairly flexible and similar.

But, as always, these are options and considerations to capture in your solution architecture.

Final thoughts - about complexity

Delivering information this way is amongst the most complex parts of the Data Vault solution. As I stated in the introduction of this paper, complexity doesn't go away just by adopting a specific methodology. Sooner or later, complexities will need to be addressed. This is not a bad thing, because it supports the auditability and flexibility in clarifying business requirements in a way that also, well, 'protects' the data teams. This complexity is needed to allow the data solution to be a tool that truly can assist the business in raising the maturity levels and address key pain points when it comes to data and information.

The good news is that, as a pattern, this can be fully automated. If you look at the full scripts, you may notice a commented-out section that says 'place business logic here'. This is to highlight that everything in the code can be automatically generated from Data Warehouse Automation software. My view is that, instead of attempting to skip complexities to keep the solution simple, we should invest the time to collaborate and simplify the management and logistics of these complexities. The ability to generate these patterns goes a long way towards keeping technical debt in check. Opting for simpler solutions will incur a penalty when re-engineering is required at some point (although the PSA will alleviate many of these issues).

This is the common "pay me now, pay me later" dynamic: if architects are familiar with the work they are postponing by neglecting to automate, at least they are making a rational decision to build a presumably "simpler" system. Instead, personally I like to work on pushing complexities to the background as commodities, so we don't have to worry about them anymore. Referring to my 'engine' paper, as 'mechanics' we may be interested to look 'under the hood' for some fine tuning every now and then. But generally speaking this can be made to 'just work', all highly automated.

Another thing to consider is that the 'behaviour of data' requires further explanation to the business. I am of the opinion that, at a certain moment – when both the back-room and front-room are fully resilient and automated, there is very little left to do for data professionals than to invest the time in 'educating the business' on the dynamics behind how data is represented. In other words, explaining (and not unimportant – getting agreement on) why in some cases results change back in time (in the case of backdating), and how this is usually triggered by systems design upstream.

In some cases, there may not even be a suitable business timeline available – another use-case where engaging the business with a view to explain the behaviour of data (as perceived in the reported results) is paramount.

It is important to note that this is not conflicting with snapshots. It is exactly the opposite. Using the LDTS filtering, it is very easy to 'lock in' what the information was at, say, the end of the month and store this for future reference. The point I want to make is that this filtering allows you to do this, as well as showing the differences between other perspectives, yet at the same time maintaining full auditability.

The approach outlined in this article provides a powerful mechanism to both satisfy one of the most common business requirements (looking at data against a timeline users are familiar with), as well as having full audit capabilities to 'go back in time' and explain why results may change slightly over time.

This Data Warehouse time machine, by virtue of the parameterised queries using LDTS filtering, can always go back in time to show the results how they once were – a powerful tool to discuss data with your business users.