Dates and times in Data Vault

'There are no best practices. Just a lot of good practices, and even more bad practices'. This is especially true when it comes to handling dates and times in Data Warehousing, including those designed based on the Data Vault methodology. Throughout its lifetime, the standards about dates and time have evolved for Data Vault – and continue to do so.

This paper aims to capture one of these discussions: the handling of dates and times in Data Vault. The starting point for explaining the evolution of handling time in Data Vault is the definition of the original Load Date / Time Stamp (LDTS). The LDTS concept was introduced in the original publication of Dan Linstedt's book 'Supercharge your Data Warehouse' and appears in the various downloadable demos / code (Informatica Powercenter) from LearnDataVault as well as in other places.

The LDTS attribute is part of every Data Vault entity and is the date / time that drives the effective dating in the Satellite/Link Satellite entities: it is 'part of the key'. The generally agreed standard (for good reason) is that the LDTS defined as the date / time that captures when data is loaded into the Data Warehouse (DWH) environment. This typically means 'stamping' the records when they are loaded into the database using a default column value (e.g. sysdatetime).

In this paper, we will show that the definition and implementation has evolved over the years and what the impact is of selecting any of the other available date / time attributes for the LDTS function.

Temporal modelling

Generally speaking, the challenges related to time-variance center around which attribute is to 'act' as effective date. There are many perspectives to cater for, of which the time the record was captured in the environment is only one. The reality is that, on a practical level and in a more traditional Relational Database Management System (RDBMS), you only assign one of these fields to be part of the key. At the same time, you aim to work towards a robust 'back-room' process that never fails to ingest data, so you are more flexible in delivering potentially different 'front-room' interpretations on the information by applying business logic.

A robust back-room tends to rely on system-driven effective dates, as opposed to dates that are delivered by source systems and hence are not under the control of the DWH team. Out of the palette of options, which one should it be? Or to put this differently, against which record should new records be compared to detect changes? This is the source of remarks about the 'shoehorning' of date / time into a database, and the comments that databases in general are inadequate at handling temporal information. Only one selected timeline is answered by the database.

This has far-reaching implications as to how information can (easily) be represented later in the delivery (i.e. Information Marts). These complexities and consequences are captured in 'temporal modelling', which is about making a conscious decision on which date / time attributes are used and for what purpose. Temporal modelling ranges from making the decision to use the default LDTS to profiling and mapping sources to a generic date / time field that accurately represents the business and its processes, and using this for reliable delivery.

Once the LDTS attribute is selected the remaining date / time attributes can be stored as contextual information. This is done by capturing the information in Satellites, which enables the ability to provide different perspectives in time at a later stage.

Examples of other contextual date / time attributes are:

- The date / time the ETL batch or workflow has *started* (across all ETL processes). This was the very first incarnation of the LDTS concept, in the early days of Data Vault.
- The date / time the ETL was *processed* (ETL process date / Job Run Date / time) – at what time was the specific ETL executed that handled the data?
- The moment of change in the source, as accurately and closest to the real change made by a user as possible but ideally computed and 'non-tamperable'. This a Change Data Capture (CDC) date / time, and can be derived from database log transactions for instance. I refer to this as the Event Date/Time (EDTS) concept.
- All kinds of date/time stamps as appear in, or are tracked by, the operational (source) system. These are generally modeled as regular context attributes (Satellite or Link-Satellite attributes).
- The moment of insertion in the database at later stages – there might be delays in ETL execution and actual inserts. An auditor might want to know when the data was loaded (and changed) in specific tables.

With Data Vault, information can be delivered in the front-room (i.e. Marts, Presentation Layer) along any of the available timelines regardless how data is stored in the back-room (Integration Layer). However, using different timelines in the Information Marts will render concepts as 'current row indicator' or 'expiry dates' void as these reflect the date / time used as the key field. In some Information Mart deliveries, there can be valid reasons to implement separate end-dating (expiry) logic based on other timelines if required.

This is one of the reasons why attributes such as the current row indicator and expiry dates are not always added to a design (performance / storage being the main other one).

In the beginning…

The LDTS in its *original* definition (in the Supercharge your Data Warehouse book) is the execution date / time for the batch (workflow) to load the (entire) Data Vault. The original design used a CONTROL_DATE mechanism (control table) where the single execution moment is recorded for the 'master' batch. This date / time is then referred to by every ETL process that is part of the batch. This design is typically geared towards a daily run, or at best at micro-batch level depending on the total runtime of the batch.

The underlying idea is that a LDTS that is created as batch date/time stamp (in the control table) is one of the few fields that are completely under control of the Data Warehouse. So, from the perspective of the Data Warehouse you reduce risk and increase robustness by decoupling dependencies from data that is outside your (direct) control. Examples of data elements that are generally outside of direct control (and therefore risky) are source dates, and to an extent even system generated date/time stamps created by the operational systems.

The way the LDTS is implemented has changed over time to enable removing batch dependencies. Using a LDTS that is issued as part of a master batch – intended to align LDTS values across all data loaded in the batch – means you are limited in your batch windows. You can't really run components of the solution independently.

However, the underlying idea of having the LDTS controlled by the Data Warehouse environment hasn't changed.

Another reason for this approach is the support for integration of data from different time zones. The LDTS concept as defined above is essentially decoupled from the time zone, as it issues a central date / time from the perspective of the Data Warehouse. This is consistent, and can be used for filtering in the delivery phase.

Interlude – event date/times as ‘real’ date / time stamps

At some point, in an attempt to remove batch dependencies, some of us (including myself) started to define the Event Date / Time (EDTS) as the new LDTS and therefore the effective date. The EDTS concept is defined as the immutable (‘non-tamperable’) moment the change is recorded. This is as close as you can get to the ‘real’ Change Data Capture event, a record changing in the source).

The exact EDTS value relies on the interface approach; the best option for this needs to be selected depending on the available interfacing options – and some are more accurate than others.

For instance, a Full Outer Join comparison to detect a data delta is as accurate (close to the real event) as the moment you run the comparison. The EDTS here is the moment of comparing the two sets (and only produces a single change per key). The EDTS you receive from a Full Outer Join comparison is relatively far away from the ‘real’ change. Similarly, a transaction-log based CDC interface in principle provides all changes at almost the exact same moment the ‘real’ change occurs – and is still system generated.

It can work, but it is risky and I advise against it. The reasoning is explained below. As a standard I recommend to maintain the LDTS as effective date / time and store the EDTS as context.

In comparison with the original Control Table implementation of the LDTS, the introduction of EDTS as effective date also intended to make sure the individual ETLs handle control tasks such as loading windows, delta selection and generally start / end times. The underlying principle is to not let ETL execution date / times to drive effective dates, which was the case in the original LDTS implementation as explained above.

If you follow this line of thought, the LDTS in its original implementation can still be incorporated, but needs to be derived using the available ETL process metadata. It can be calculated as a point-in-time view on the available information from the perspective of the ETL Batch execution.

By using the EDTS as effective date you get a closer representation of changes as they have occurred in the operational systems, and which is still not influenced by users – the EDTS is not a source attribute. This approach does require (even more) focus on adequate interfacing design, but levels the playing field for all upstream ETL logic. It is reliable and predictable: it doesn’t matter when you run your ETL. Regardless whether you run your processes in the morning or afternoon, the results will always be the same.

There are also downsides to this:

- Depending on your technical architecture and (types of) interfaces you may run into time zone issues. If you use a replicated database (with CDC enabled) or similar in your own environment it’s not an issue. But if your CDC database is in another time zone, or the server changes time zones, you may get key violation issues and generally experience incorrect order of recorded time. This may even happen in daylight saving time scenarios if you’re particularly unlucky.
- Due to this, the above approach is usually not sustainable in a decentral and global Data Vault solution. To an extent, this can be handled using concepts such as applying UTC (Coordinated Central Time) – more on this later. But this is *only* achievable if you can sufficiently control the interfacing – which is not always the case.
- Similar to the LDTS, you *still* need to reorder data along other (business timelines) for delivery. The EDTS cannot be considered a business timeline.

In general, applying time zone business logic into your initial interfacing ETL is a very risky approach, and is recommended to be avoided. This is the reason why the LDTS was defined in its original form as the 'arrival date / time' in the first place.

LDTS restyled for DV2.0

As part of updates and improvements in the Data Vault concepts as part of 'Data Vault 2.0' Dan Linstedt has suggested a revised definition of LDTS: the date / time the record was received (inserted by) the database. This is different from the original incarnation where the LDTS applied to all records loaded by an overall batch – in DV2.0 records are time-stamped individual whenever they become available. No more batch processes.

In terms of solution design, this means the date / time is recorded when a record was received by the Staging Area or the Data Vault (or both) although the official standard is to set the LDTS in the Data Vault. However, a common approach is to 'lock in' this date / time in the Staging Area, so information can be propagated from there in a deterministic way (a requirement for the Virtual Data Warehouse).

Often multiple Data Vault tables are populated from a single Staging Area table, and propagating the LDTS from the Staging Area guarantees deterministic and synchronised date/times. However, this may not always be possible for instance in the case of near real-time ETL.

Capturing the LDTS in the Staging Area also allows for a truncate and rebuild of the entire Data Vault if information is archived into a Persistent Staging Area. While this concept is not part of the Data Vault definitions it is widely used to support refactoring and virtualisation. In any case, this new definition means the LDTS is not an ETL driven date/time or control date, and as an 'automatic/triggered' attribute this would support the processing of multiple changes for the same natural key (in a single dataset).

It is important to realise that this definition of the LDTS will always make the record unique, which especially has a large impact if the solution is designed to load into the Data Vault Satellites directly. Because uniqueness is not handled on database level (by the Primary Key) you may be required to define additional row condensing mechanisms such as checksum comparison (full row hashes) to prevent from reloading the same context data more than once.

Without these checks in place even accidentally rerunning the same ETL twice might result in duplicate data. It is out of scope of this paper, but can be summarised as the implementation of a more holistic Change Data Capture approach between the source system and the Data Vault.

These various considerations have been listed below.

If the LDTS is 'locked' in the Staging Area (before loading this into the Data Vault), the following applies:

- You can truncate and rebuild your Data Vault from an archive (Persistent Staging Area, Hadoop/HDFS file delta), and are able to reproduce this in the exact same way (e.g. representing the same LDTS).
- You can prevent loading multiple loads (duplicates) as outlined above by comparing on the LDTS you set in the Staging Area. If you already lock in the date/time in the Staging Area, you can just check on this value (and the natural key) to prevent reloads.
- You *also* may want to log the moment the record is inserted into the Data Vault.
- In full near real-time interfaces the Staging Area may be skipped, which obviously means the LDTS cannot be set on the way in to Staging. There are various ways to work around this if required, depending on the architecture (virtualisation being one of them).
- The same LDTS (for a given Staging Area table) can be consistently applied in all upstream tables that are sourced from that Staging Area table.

If the LDTS is 'locked' in the Data Vault (Integration Layer), the following applies:

- If you truncate and rebuild your Data Vault from an archive, your effective dates will be different each time you do so.
- Preventing loading duplicates as outlined above requires additional concepts as part of the row condensing / CDC. You cannot depend on the LDTS to prevent reruns, so you need additional information to prevent records from being reloaded. In normal situations checksum mechanisms can be applied to compare information between new (Staging) and existing (Data Vault) information. This is typically executed against the most recent (no pun intended) record in the Satellite.
- However, the above only works when a single key delta is processed. When (re)loading *multiple changes per natural key* in a single pass, every record also needs to be compared against… something. This may be the Event Date/Time, or a (system) ROW ID, or both. This can also be incorporated into the row condensing concept.
- Similar to the previous bullet, if you reload historical data against an existing historical data (e.g. Satellite) set you need to be able to do a point-in-time comparison, for which you need another date / time than the LDTS. You may want to do this from time to time as a full compare to detect any gaps in interfacing.
- Using the pattern generally allows you to omit Staging (and Persistent Staging) areas in your architecture.

These are themselves not new concepts; but it is interesting to consider that you still need another date / time than the LDTS in the Satellite (checksum) if you need to make a comparison for late arriving data, or to compare a (full) historical data set against an existing Data Warehouse. You can use the EDTS for this.

In practical terms, in addition to catering for the various date / time stamps outlined in the introduction it is worth considering adding the EDTS. The EDTS is not a 'source attribute' – it does not appear in the source table but needs to be provided by the interface. Having the EDTS as additional context provides you with a second computed timeline (second to the LDTS) to support Data Warehouse function you may want to leverage (i.e. the gap detections). And, if nothing else, it is another available timeline you can use to determine if information has truly changed.

A simple example is the scenario that you move back to your original address. The record in itself may completely identical to a record that was processed earlier – except for the EDTS. If you omit this in your checksum or comparison, the record would be discarded and the change would be missed. This will not cause issues when only comparing a single record against the most recent record, but in the event of processing multiple changes and/or reloads/merges of historical data in a single pass this needs to be considered.

But we're still not there yet. By applying EDTS to the DV2.0 LDTS concept the requirement to load multiple changes per key, as well as the requirement to reload your Data Vault from an archive in exactly the same way are completely satisfied. The risk of exposure to time zones issues is still there if you use anything beyond the LDTS – but the impacts are much smaller if this risk materialises.

The DV2.0 LDTS approach may requires additional metadata to truly make the record unique, for instance when changes arrive too fast for the time-stamping to keep up. The 'additional thing' that makes the record unique needs to be defined on a per-case (interface) basis. Options that come to mind are the delivery date on the file system in combination with the Event Date/Time for flat files. But a source ROW ID, or a database log sequence number can also be added – or a combination of both.

Anything that makes the record unique, and what is available (and ideally immutable) again depends on the interface options, and options depend on specific situations and the system landscape. There is no one-size-fits-all solution, although in my personal opinion incorporating the LDTS to be set in the Staging Area is the best solution.

This is assuming you are not directly incorporating near real-time into your Data Vault of course.

An additional issue that might occur is that, if you use a trigger-based mechanism to create this type of LDTS on database level, the system may not be able to cope well with handling large volumes of data in one go (for instance for an initial load). This is usually OK, but performance may vary depending on the technical architecture.

Handling time zones

As a centralized date / time value, controlled by the Data Warehouse team, it helps to record the LDTS in UTC. If required, the UTC value can be converted to any local time zones for auditing purposes. This is relatively easy because you only have to conform with a single time zone: your local one, where the Data Warehouse runs. Another key function for the LDTS is filtering, and for this the time zone doesn't really matter.

At this stage it is worth considering storing contextual date / time information you receive from other systems in a data type that can capture the UTC offset as well (i.e. datetimeoffset), or store the offset (or time zone identifier) separately if no suitable data type is available. This can be important when dealing with data that originates from other time zones (i.e. international transactions), as it makes it easier to cast the values in different time zones. As an alternative you could also transform incoming data to UTC (while also storing the relative offset to the original time zone).

Both can work, as long as you are consistent – I usually pick the first option and store the received values in their local time zone including the UTC offset. As an example, this can be implemented using the *datetimeoffset* data type.

An example is when you need to combine data that has a time-critical element such as reporting on Service Level Agreements. To understand if, say, tasks were completed within office hours, you need to know if the reported times are actually reflecting reality for that time zone – which should also take into account time zone shifts (i.e. daylight savings). The underlying reason is that the UTC offset is not fixed. Rather, it changes throughout the year and may follow different rules per time zone.

It is these time zone shifts that usually cause problems, and it has caught me off guard while working with the EDTS in the past. When, from time to time, the time zone shifts the interpretation of the 'before and after' changes and causes leaps in time.

In the end you should be able to deliver information according to the timeline that the business user is familiar with, as explained in the pattern for Data Mart delivery. This means that you need to capture all date / time information that is associated with the event separately from the time the information arrives in the Data Warehouse. To accurately represent this information across time zones you will have to capture the local time including the UTC offset. Due to the time zone shifts, this may mean that some transactions may appear 'twice' in the history – which is correct in the case where daylight savings adjustments are applied. The only way to interpret this information is if you include the time zone shift in your query, i.e. specifying if the reported time was before or after the time zone adjustment.

Depending on the platform this can be remediated in various ways. SQL Server 2016 for example now has a new way of addressing some of these challenges with the 'AT TIME ZONE', which also takes into account shifts like daylight savings (by and large – there are some exceptions).

Final thoughts

There are many considerations to take into account; which all form part of the discussion about 'temporal design'. It has always been an alternative to model time differently for each Satellite, even outside of the concepts explained here. The temporal modeling issue comes down to figuring out exactly what will make a record unique for that particular source / interface combination.

The real underlying discussion is balancing out standards and architectures versus per-interface design decisions and reliability. In other words, how many patterns do we want to support and can we manage the risk? Should we have one pattern for batch processing and a separate one for near real-time processing? This is where opinions vary, and I'm of the opinion that it is OK to select different interface patterns to support generic upstream processing.

I tend to use the Staging and Persistent Staging Area to deliver consistency upstream – and only use LDTS for filtering.

The best outcome is that you have control over the interface as with a transaction log-based Change Data Capture solution in the Data Warehouse environment. This way you can be very specific and even rely on the Event Date/Time. But as discussed in the previous section, in most other scenarios you need to add discriminating attributes to the key (of the Satellite), which comes back to information modeling and a fairly deep understanding of what each date / time attribute means as well as if and how it can be used for this purpose.