Analysing the breadcrumbs

Data can be described as the breadcrumbs of human activity, recorded in our information systems. It is the by-product of (business) processes - which themselves are organised (orchestrated) tasks - that lead to, or produce, a specific outcome or set of outcomes. The tasks that form the business processes generate data.

When this data is collected and integrated from disparate systems into a Data Warehouse the intent often is to ultimately expose the data in a way that is conformed towards some form of use-case or requirement. Invariably, this requires data to be transformed into something different. You could say that during this process context, interpretation, is applied and data is subsequently 'becomes' information.

Regardless of the methodology used this complexity needs to be managed somewhere. In Data Vault this application of logic (i.e. the transformation) in the form of applying business rules is done on the way *out* of the system. Towards the 'Data Marts'.

This is intended to be a fairly iterative approach, for the simple reason that often it takes time and effort for these business rules to fully crystallise. In many cases the exact required logic cannot be provided upfront in a way that provides the expected result. There are various reasons for this, including the level of 'business understanding' of 'how data works' in terms of back- and future dating (applying changes to data at a different point in time), differences in expectation about the business processes that produce the data (documentation versus reality) and data quality issues in general.

Issues such as these lead to a situation that even experienced Subject Matter Experts (SMEs) usually need various iterations to truly match data to processes and expectations. In other words: it is very hard to accurately define requirements at the start of a data initiative. At its worst, understanding data can be a very forensic exercise where recorded data may even seem completely meaningless at first, especially when there are large gaps in the (documentation of) business processes.

You could argue that the more 'technical debt' there is in business processes, the more pressure will be placed on the Data Warehouse to absorb complexity. Conversely, the better business processes are designed, implemented and documented the easier it will be for Data Warehouses to produce expected results. As data professionals it pays off to be mindful of this dynamic.

This comes back to a point I often make in papers and presentations: I recommend to treat the Data Warehouse as a facilitator, a mechanism, to explain what is really happening in the business by virtue of presenting the facts (raw data) and using this to drive changes in the wider landscape. This is opposed to architecting the solution to deliver the Single Version of the Truth and dealing with what is basically the organisation's technical debt through business rules implemented in the Data Warehouse environment.

A similar point can be made regarding multiple potentially competing definitions. Someone told me 'I can't believe we have five different ways to calculate and report on revenue'. My response is that, in my view, this is something that is certainly far from ideal but is not the job of a Data Warehouse to resolve. Rather, that is for Data Governance to address.

The Data Warehouse can provide tremendous value by *allowing* these multiple definitions to co-exists, as long as these can always be traced back to the original facts (raw data) – and an overall (governance) approach exists to resolve this at a business level.

Following the 'virtual Data Warehouse' mindset; as soon as the various teams have agreed on a consistent way forward under the supervision and guidance of Data Governance, the Data Warehouse can be refactored to combine these various perspectives into one agreed standard.

Thank you for your business requirements.

A Single Version of the Truth can be achieved this way, over time. But this is achieved by architecting the Data Warehouse as facilitator, and by enabling iterative development supported by Data Governance – not as a one-off implementation of a single definition.

This introduction highlights the high-level considerations that are part of business logic implementation. Should the Data Warehouse absorb the existing issues in the organisation's business processes, this technical debt? Are there ways processes can be adjusted over time, when facts are provided through the Data Warehouse? The more data and processes are organised better, the less complexity is required in the Data Warehouse – the less business rules are required.

Having said this, in the most cases there usually still is a need to implement logic in the Data Warehouse to deliver the end result consumers seek. To achieve this there is great benefit in exposing data early (in a 'raw' state) to the intended consumers. Exposing raw data in a structure suited for consumption both allows early feedback and displays progress. In Data Vault this is referred to as initially delivering a 'Data' Mart and gradually transforming this into an 'Information' Mart through the (iterative) application of business logic together with business SMEs.

Test and learn - continue to clarify requirements until they meet the consumer's needs.

Flavours of business logic

For the purpose of explaining how and where this kind of business logic is applied and to avoid any potential ambiguity I will stick to the terminology of 'Mart' (for any kind of information delivery) and 'business logic' for any kind of implementation of transformations to support business rules.

So, let's talk about 'business logic' and what it really is. Most people are familiar with notion of applying business logic as transformations implemented using ETL software or coding frameworks such as C#, Python, or SQL. For the sake of brevity, I will refer to these collectively as 'ETL processes' - including various implementation nuances of this including ELT and LETS.

Usually this business logic is documented in plain English as the result of 'requirements' conversations, and subsequently translated to the code that corresponds to the target environment. Transformations created using techniques like these can cover a wide scope of requirements, including but not limited to aggregations, derivations, classification and approximations (i.e. probabilistic 'fuzzy' matching etc.).

Using the Data Vault methodology, we will need to do this as well. However, before we do it is worthwhile to take a step back and consider the role of interpreting data in the end-to-end architecture. Interpreting data can be seen, in a wider context, as everything that is required to morph data into the form that is required for it to be accepted by consumers.

If we look at business logic this way there is a lot that is already covered by a hybrid Data Warehouse architecture (such as Data Vault) and standard patterns - by separating concerns.

Architecture & data models: separating concerns to simplify logic requirements

There is much that a good architecture can do to simplify what business logic is really required. Hybrid modelling approaches, such as Data Vault, are well equipped to support this 'out of the box'.

For comparison; solution designs that require application of business logic in order to allow data to be stored in the Data Warehouse (i.e. the 'Kimball' approach or a 3NF model) tend to require relatively complex ETL processes.

The reason for this is that a lot of things need to be done in a single ETL step. Activity such as (Data Warehouse) key distribution, handling time-variance (historisation), setting up (and transforming to) structure and hierarchy, column mappings, combining multiple data sets (joins), handling granularity issues and of course transformations based on business logic.

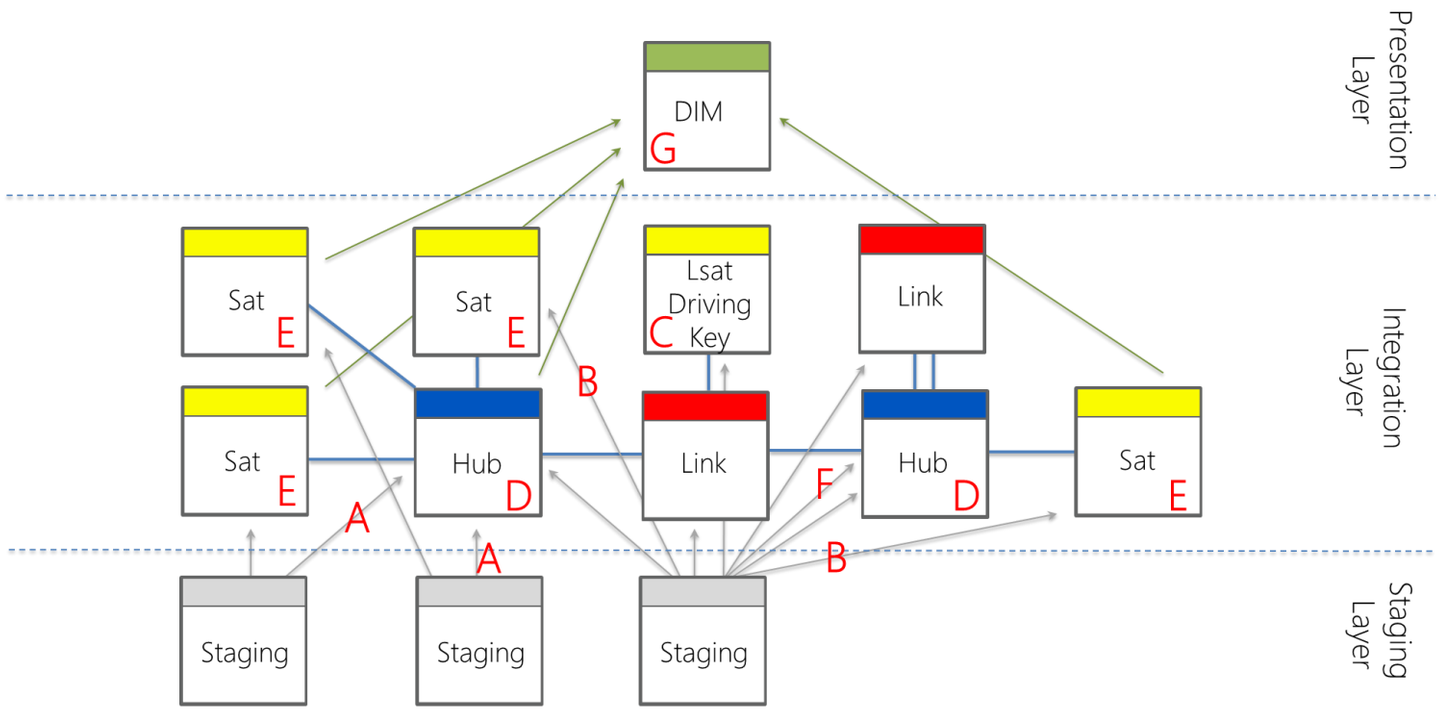
All of this is required for data to meet the constraints of the target model: data is 'judged on the way in'. This model is often a representation of an ideal, and in many cases means that data needs to undergo intrinsic changes in order to be loaded - some would say 'shoe-horned'. These approaches also require exceptions to be handled upfront, which may lead to some data being delayed in its arrival into the Data Warehouse.

This is not necessarily a bad approach, but as an architect it's important to be aware of the various pros and cons associated with these solution designs.

While there are various variations on solution design when it comes to any methodology, for Data Vault there is a clear distinction between the 'back-room' where data can be loaded quickly, efficiently and reliable and the 'front-room' where the interpretation can be done. As a concept, pushing the application of business logic to the front room means that you don't have to deal with exception handling in the back room, which further increases reliability. Also, the front room is closer to the business (consumers) which increases speed of delivery and iteration when working with SMEs.

If we look at this architecture in detail, you could argue that there are many steps towards the interpretation of data that are already natively available. Steps that could be classified as part of business logic in a more traditional approach as explained above. The separation of concerns brings business logic back to the essence of the transformations itself, and is not confused with the various Data Warehouse activities and tasks.

The following diagram provides an overview of what is already available 'out of the box' in a vanilla implementation:



A) Passive integration; by mapping (the Business Keys of) various data sources to the same business concept they 'become' the same thing.

B) Column-mapping; mapping source attribute names to something more suitable is possible, under the condition that no (destructive) transformations are applied. It is of course possible to do this mapping later on in the Mart delivery, or store this separately in a Business Data Vault object (covered later in this paper). But it is generally easier to apply column mapping early on to simplify the process of generating a Mart.

C) End-dating of relationships. When a relationship changes, an old (previous) version may need to be logically expired. Operational systems sometimes are not always able to provide this information directly, and in these cases concepts such as Driving Keys (or alternatives thereof) can be configured to detect and manage this. I personally consider this business logic, since an interpretation is required as to which element of the relationship triggers the end-dating (i.e. the determination of the Driving Key). This is a specific (modular) function that can be completely separated from the implementation of business logic in the narrower context of transformation.

D) Key distribution; through assigning a Data Warehouse key (i.e. sequence value, hash) as part of the Hub loading processes. This then becomes the integration point, the key everything can be linked to.

E) Handling time-variance; Data Vault has an elegant way of dealing with dates and times. The Load Date Time Stamp (LDTS) concept records the time of arrival in the Data Warehouse environment and 'acts' as effective date. This means that changes over time are recorded in the order of arrival and any other date / time information including effective periods or business reporting timelines are stored as context.

   As a back-room technique - a value that is managed by the Data Warehouse - the LDTS ensures historisation without impacting reliability. It cannot be subject to data quality issues that could otherwise impact the loading processes.

   Managing attribute selection scope also falls in this category. The more attributes you use, the more rows you will have in a typical time-variant data set. Similarly, the fewer attributes you select the smaller the row set becomes. This is the (intended) effect of record (row) condensing - a critical component to handle time-variant data (i.e. Satellites, Marts, PSA). See '[when is a change a change]' for details on how this can be implemented.

F) Similarity and recursiveness; by using Same-As-Link and Hierarchical Link structures. This is a somewhat ambiguous area because in some cases it is perfectly possible to load data directly from operational systems into these structures, while in other cases true business logic has to be applied to detect similarities or otherwise draw relationships that are not directly represented in the source data. Examples of this are probabilistic (fuzzy) matching or decision trees. Assuming data relationships are available in the source, these can be loaded as-is.

G) Delivery timelines. While in the Data Vault the LDTS is used as effective date, for consumers a more meaningful timeline for reporting is usually required. This means that data needs to be re-ordered according to the intended delivery timeline. The correct selection of the timeline for delivery is a front-room decision. This interpretation is a specific step in delivering Marts, possibly including PIT tables. This mechanism is covered in detail in the paper '[a pattern for Data Mart delivery]'.

These are all parts of the overall design that assist in making data 'fit for purpose'. The message here is that a significant amount of preparation work towards preparing data for consumption is already covered, which only leaves the true business logic to be applied. This is the intended outcome of the architecture: to determine specific areas where only the essence of the business logic can be implemented - in a contained and concise way.

In other words: having some of the above components in place means that the scope of what *actually* needs to be handled by business logic becomes relatively succinct. The above concepts are embedded in the core Data Vault patterns and do not require any specific interpretation other than adequate data modelling and implementation of the Data Vault.

Therefore, it is fair to say that a lot of business logic implementation can be avoided simply by having a good solution design.

Areas of business logic application

Now we know what functionality the standard patterns already provide we can further limit the scope of business logic to pure *transformations*. Generally speaking, in a Data Vault solution there are two options where these can be applied:

- On the way to the Mart
- Within the Data Vault as derived information (i.e. Business Data Vault)

The most straightforward way (conceptually at least) is applying transformations when loading the Marts. Once you join the various Hubs, Links and Satellites together for the required (base) attributes you can apply the business logic that delivers the right business outcome. This is a very straightforward use-case and is easy to implement.

This approach works well if the transformation is very specific, i.e. only relevant for a single Mart. You may find however that certain interpretations may be required across multiple Marts. It is a development best practice to embed specific code (logic!) in a single place. If you don't, the same transformation rule may be implemented in multiple locations ('copy-and-paste programming'). When you find that certain transformations need to be used in multiple places it may be worth to consider a more reusable implementation.

In these cases, it is worthwhile to embed the logic in the Data Vault itself so that it can be reused many times towards different Data Marts. This is the standard Data Vault solution and is referred to as the 'Business Data Vault'.

This is the second option, which introduces the concept of the 'raw' Data Vault and the 'Business' Data Vault. I personally prefer to call these 'base' and 'derived' but for consistency's sake I'll stick to the original names.

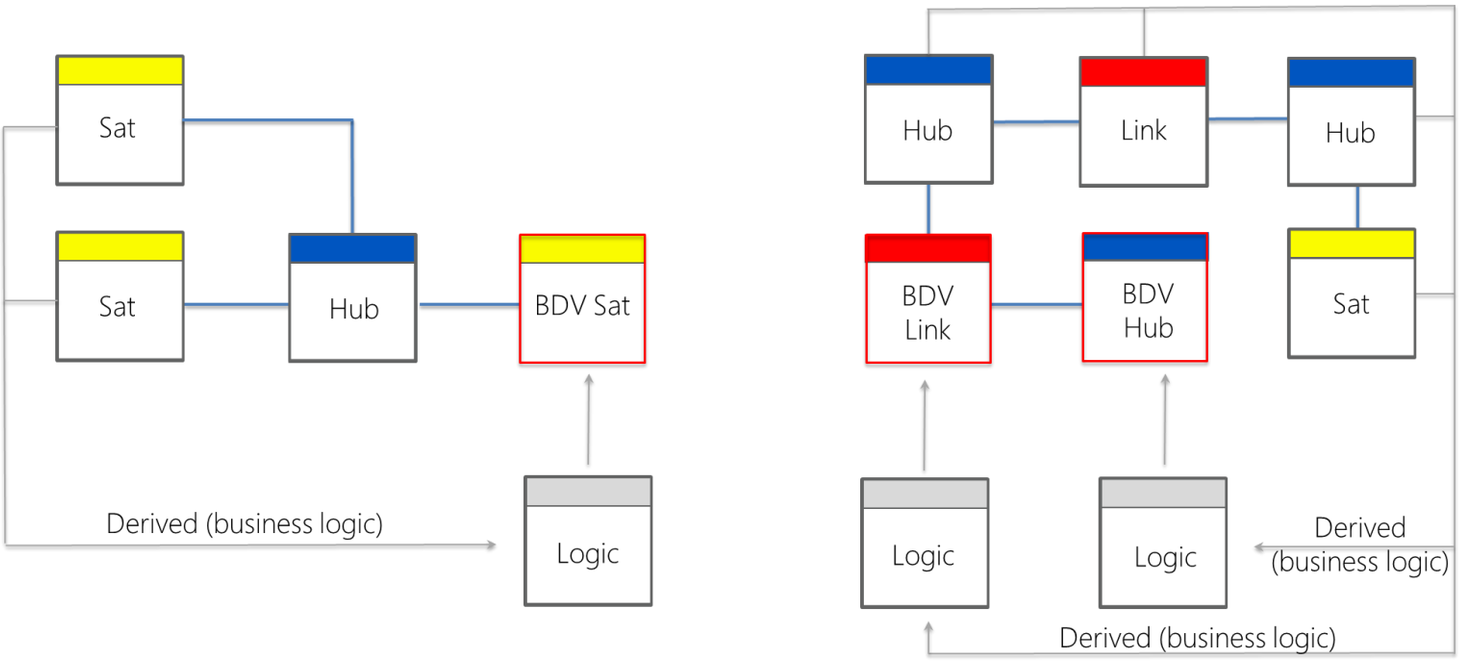
Different people can have a different understanding of how 'Raw' and 'Business' Data Vault works. I would like to clarify that these are not 'layers' in the architecture and not physically separate parts of a Data Vault. The Business Data Vault is an *extension* to the Data Vault where specific business logic is applied.

In others words: if the source of a Data Vault table is the operational ('feeding') system then that table is classified as a Raw Data Vault table. In alignment with the core Data Vault principles, there will be no destructive transformation applied - transformations that irreversibly change the content.

On the other hand, if a Data Vault table is loaded through the implementation of a business rule it is classified as a Business Data Vault table. Business Data Vault objects are in principle not loaded from the operational system, but from Raw Data Vault tables - otherwise the audibility and ability to refactor will be lost.

Combined, the 'Raw' and 'Business' areas *are* 'the Data Vault'. It is a single model, the Integration Layer, that contains the original data that describes business objects and any generic interpretations thereof. Other than how they are loaded the Raw and Business Data Vault tables are similar in structure and design. They are all part of the same Data Vault model, and can include the typical Hub, Link and Satellite archetypes.

This is displayed in the diagram below.



From a technical perspective I would like to make a note that the concept of having Business Data Vault load from Raw Data Vault tables does not conflict with the fully parallel loading concept. This is out of scope for this paper, but explained in the paper covering '[referential integrity and orchestration in a fully parallel environment]'.

As mentioned earlier, Raw and Business Data Vault are not separate layers and there is no need to 'pass through a Business Data Vault' to be able to load Marts. This is important to state, as I've seen various examples where data has been copied from a Raw to a Business Data Vault table without any transformations applied, just because every step throughout the layers had to be followed.

As always, don't constrain yourself too much when making these choices. It's OK to refactor.

Translating intent to code

We have now narrowed the scope down to actual transformations, the business logic itself. A best practice is to make sure business logic is recorded and classified in a repository. I recommend the following:

- Uniquely number everything
- Store business logic (rules) in a repository
- Apply version control – record changes of logic over time
- Use IDs / pointers for the business rules in code; create a link between the logic and where it is applied in code

The idea is to make sure you have a managed list of rules which is directly linked to the implementation. An example of a repository is provided here:

Rule ID	Business Rule
2	Define Order Of Dependents
6	Determine Policy Holder (Owner of Cover)

Row ID	Rule ID	Effective DT	Expiry DT	Audit DT	Reasoning	Implementation Pseudo Code
1523	2	1900-01-01	2019-01-19	2018-07-10	Select the order of creating defines the order as this information is not captured.	Rank Customers on Policy by [MODIFIED DATE].
1524	2	2019-01-20	9999-12-31	2019-01-20	For the Travel policies the order of creating defines the order as this information is not captured. For Roadside assistance this is captured however and can be used directly.	If the [Plan_Code] is 'Travel' then rank Customers on Policy by [MODIFIED DATE]. If the [Plan_Code] is 'Roadside' then select the [DEPENDENT_NR] value.
1525	6	1900-01-01	9999-12-31	2018-01-15	If the order of the customer on the policy is the first, then this is the owner of cover	Rank the order of customers on a policy using [Define Order of Dependents] and select the Rank = 1.

Using Domain Specific Languages (DSLs) is a good way to streamline how these requirements are documented, and provides further avenues for automation. A DSL (sometimes called a 'mini language') is a custom language designed for a specific purpose or application. By using conventions and agreed words a DSL can be used to provide standard structure to requirements, which can be (developed to be) understood by IT systems.

Personally, I'm interested in further exploring how linguistic patterns can be used to 'compile' structured specifications into code. For example, using Fact Oriented approaches (i.e. ORM, FCO-IM, FAML).

In any case, the identifiers of the logic can be added to ETL processes to create the link between the code and the documentation. In its simplest form business logic can be implemented as views using plain SQL, but many forms of delivery are possible including web services, business rule engines and any available ETL components (tool dependent).

In all cases it is worthwhile to record the rule to the code where the rule is implemented.

The logic bitmap

A personal favour of mine to implement the link between the documentation and the code is by using a bitmap value - and I call this the 'logic bitmap'. This is a reuse of the 'error bitmap' concept written about earlier (more than 10 years ago!).

The logic bitmap works the same way: every rule is uniquely identified and assigned a 'bit position' which is a binary identifier. When rules are used, the corresponding bit positions are selected and summed. The eventual value that is recorded against data is the sum of these bit positions.

This idea is illustrated with an example below:

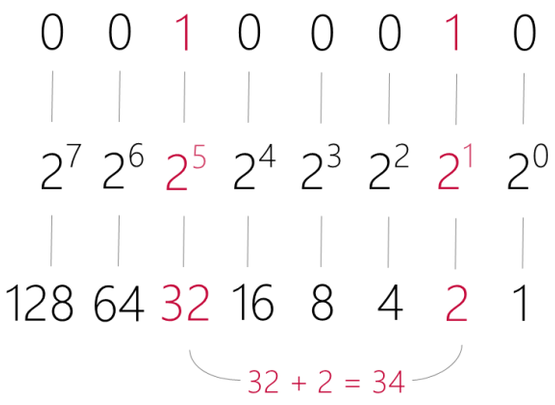
Scenario	Description	Bit position
1	Add GST	1
2	Define Order Of Dependents	2
3	Calculate Earned	4
4	Calculate Written	8
5	Invalid phone number	16
6	Determine Policy Holder (Owner of Cover)	32
7	Standardise Address	64
8	Remove GST	128

In this example, rule number 2 and 6 are applied as transformations in data. The corresponding bit positions are 2 and 32, which means the total value of 34 (2 + 32) is calculated as the value that represents both 'active' rules.

This integer value of 34 can be represented as a binary (bit-wise) value of 001000010 – hence the bitmap concept. Read from right to left! Every zero and one shows which if a rule has been used for a specified position.

This is an elegant way to link multiple scenarios (rules as records) to a single record in the data platform. Many database engines have built-in support for 'bitwise joins' to query this information directly without any further complexity.

They can do the below translation automatically.

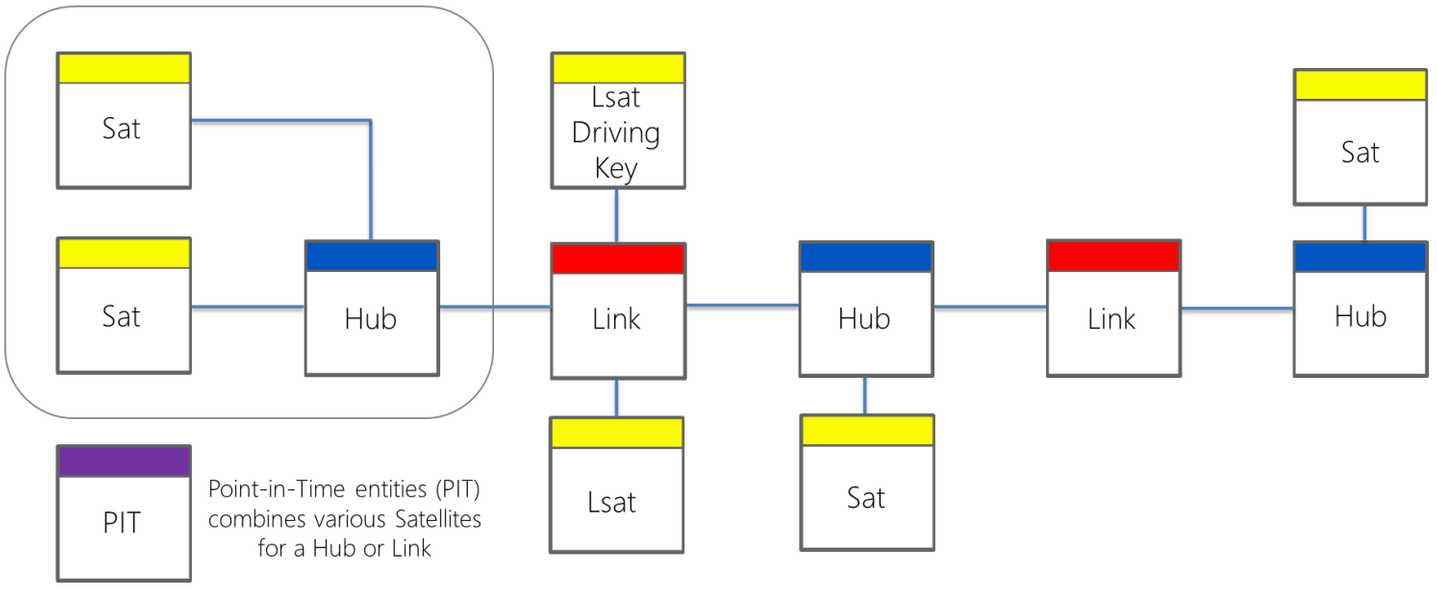


Performance enhancers

Data Vault methodology provides additional optional constructs that can be used to facilitate performance optimisation, the most common ones being the 'Point-In-Time' (PIT) table and the Bridge tables. Both are primarily intended to boost performance in some cases but can be used to house some business logic as well (more on this later).

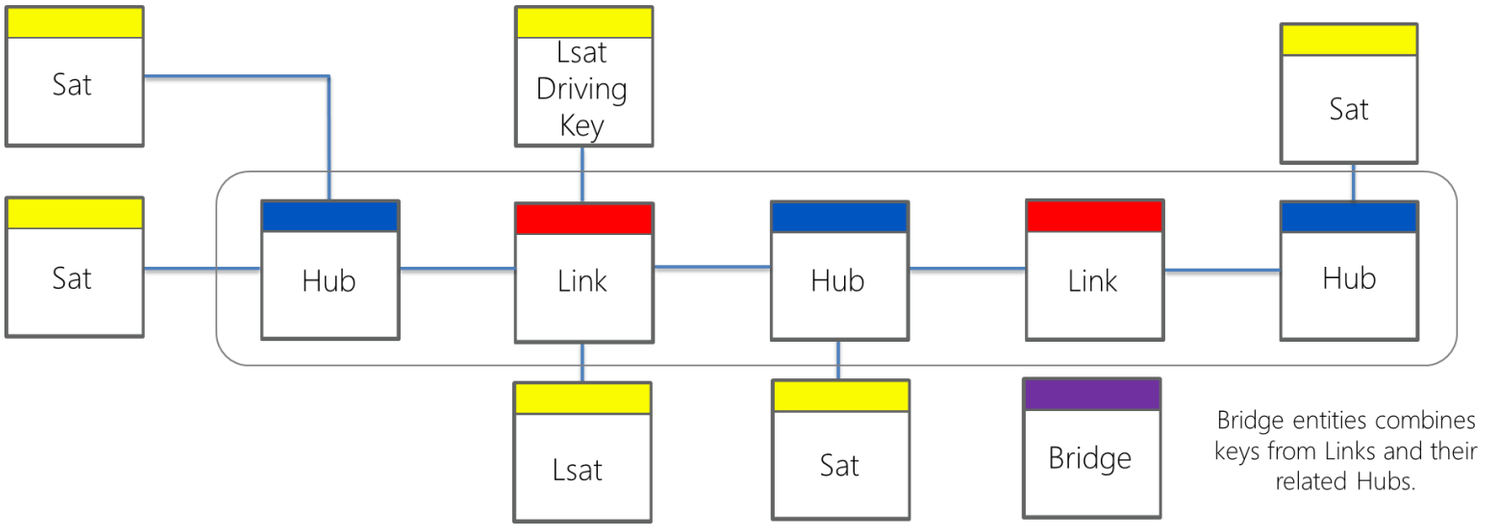
For this reason, and the fact that they can be reproduced (refactored) deterministically, PIT and Bridge tables can be classified as being part of 'Business Data Vault'.

The below diagram explains what a PIT table is: the combined record set from various time-variant tables. This can be aggregated for a certain point in time (hence the name) and usually contains the keys and date / time required to join in the required context.



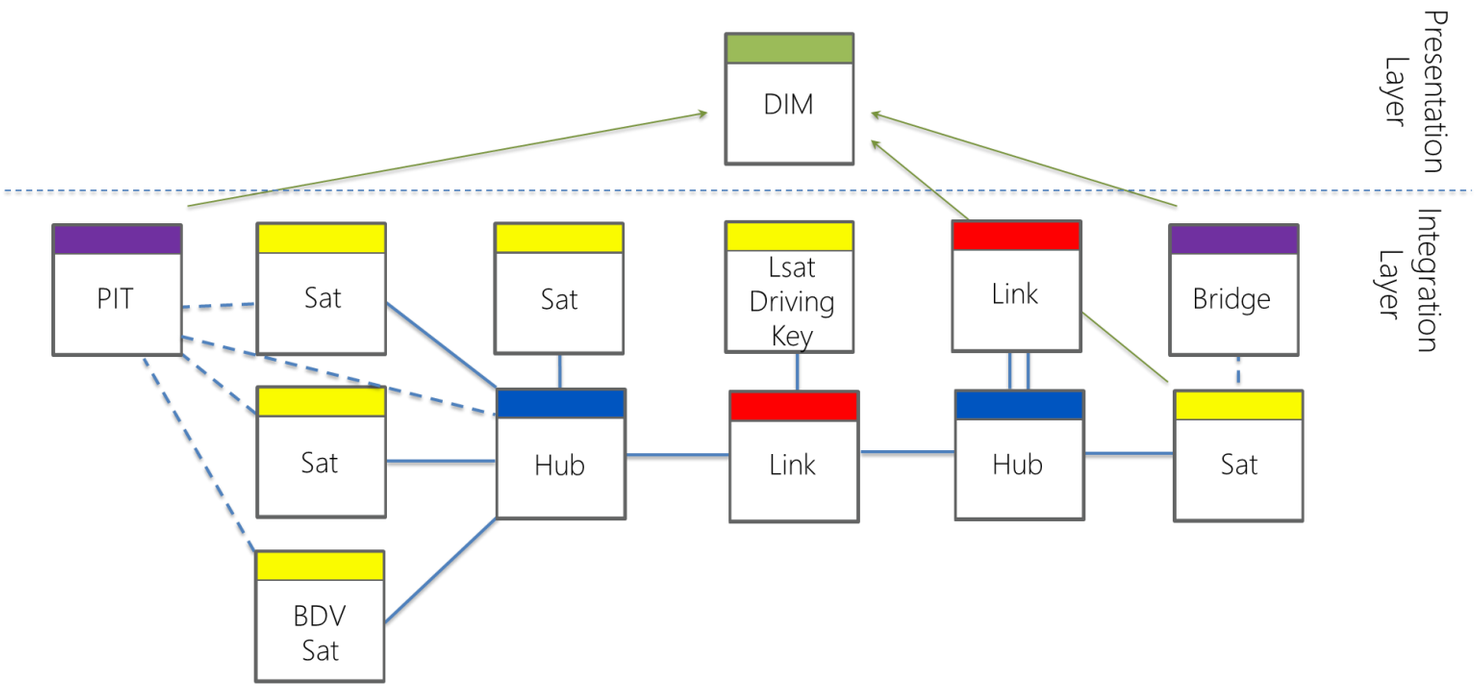
It is possible to add some attributes to a PIT table, but it's important to remember the underlying reasons to develop this entity type: performance. The patterns for loading a PIT table or a Mart are pretty much identical, and the more attributes you add the more the end result resembles the Mart object (a Dimension table for example). It is therefore critical to find the balance for the optimal performance gains that warrant the addition of a PIT table.

Bridge tables are similar in approach, but focus on joining together Hubs, Links with their Business Keys as semi-constructs to streamline loading processes – if required. This is displayed in the diagram below.



Bringing it all together

Regardless if Business Data Vault objects are used or not, the existing Mart pattern can be used to deliver information in the same way. The only difference is that 'raw' Satellites are not used in favour of their derived counterparts. The full Mart pattern can leverage all Raw and Business Data Vault objects, including PIT and Bridge tables. This is displayed in the next diagram.



The end solution can also reuse the existing loading paradigms, including continuous loading. This makes it possible to load interpretations in parallel of regular Data Vault objects – and load the Marts when the data is ready!