Business Vault & Activity Schema

Let’s discuss **event-driven architecture** (EDA), EDA is the basis of asynchronous communication between services in a microservices architecture. Kafka is *the* well-known technology that supports microservice communication as append-only log structures (but not the only one). A log is an immutable file structure where producers publish events to topics (logs) and consumers subscribe to their topics of interest. Events are efficient, fault-tolerant when parallelised through partition replication (which also improves performance), guarantee exactly once semantics, read in the order it is written and supports offset tokens for consumers to pick up where they left off.
When it comes to running analytics based on EDA the data may not be provided in the ideal form for running analytics efficiently using SQL, and this is where the Activity Schema comes to the fore. Developed as a simple table structure called a *stream* with the same **ten** columns no matter what the stream is describing, the only details that change are

- the **entities** the stream is describing
- the **activities** *you* choose to model and therefore derive the analytics you’re after

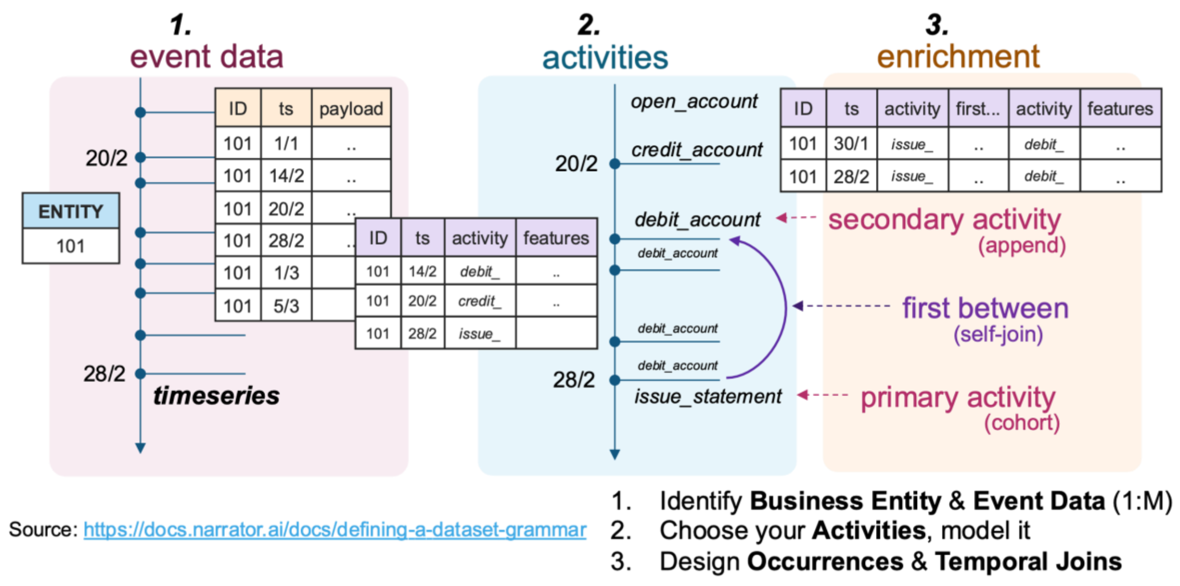
What also makes this modelling paradigm attractive is that the inventors claim that there are a set of **twelve** templated enrichments (called *temporal joins*) that almost *all* analytics performed on timeseries data that this data modelling framework supports.

How does it work?

Event data can be subscribed to or landed as batch files and the form they arrive in is typically semi-structured as JSON or XML and serialised using Protobuf or AVRO protocols or transported as just plain text files. The data is raw and will include event data about a business entity and interactions between business entities as key-value pairs to which there can be any number of properties (even nested properties) about the event.
Because this data is raw it *might* not be in the ideal form to perform templated joins on the data to derive analytical value. Thus, the first step of **Activity Schema** is to model the data into something usable and in the form we can apply enrichment and intelligence to. To build an Activity Schema we follow this documented three-step process,

1. Choose and design your activities of interest
2. Find relevant raw data source(s)
3. Write a SQL query to transform events into each activity

The result of step 3 is in the form of those ten columns we discussed earlier.


1. event data
2. activities
3. enrichment
ENTITY
101
20/2
28/2
timeseries
open_account
credit_account
debit_account
debit_account
debit_account
debit_account
issue_statement
secondary activity (append)
first between (self-join)
primary activity (cohort)
Source: https://docs.narrator.ai/docs/defining-a-dataset-grammar
1. Identify Business Entity & Event Data (1:M)
2. Choose your Activities, model it
3. Design Occurrences & Temporal Joins

ID	ts	payload
101	1/1	..
101	14/2	..
101	20/2	..
101	28/2	..
101	1/3	
101	5/3	

ID	ts	activity	features
101	14/2	debit_	..
101	20/2	credit_	..
101	28/2	issue_	

ID	ts	activity	first...	activity	features
101	30/1	issue_	..	debit_	..
101	28/2	issue_	..	debit_	..

Figure 1 Event Stream --> Activity Stream --> Dimensional Enrichment

Now that we have an activity stream, we can enrich that data with dimensional data for analytics where we need them (facts and dimensions) and with the twelve templated self-joins. How *could* the above then be reused for a Data Vault? It can *(and should)* if you're building a data vault for event data and Snowflake has the features to stream the data through a data vault *incrementally.*

Let's go!

Activity Schema in the Data Vault

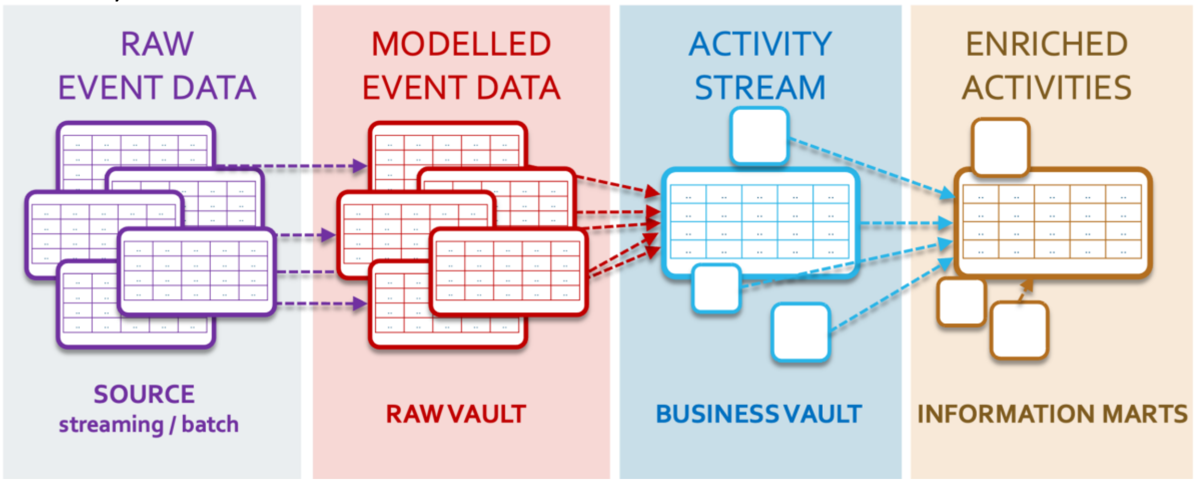


Figure 2 High-level flow

As we have described, event data is tracking business entities and the interactions between those business entities and other business entities. All other data in that payload is tracking state data about those business entities and business entity interactions. From this basic premise we have a raw vault, as:

- **Hub table**(s) recording the business entity business key(s)
- **Link table**(s) recording the unit of work, transaction, or relationship between business entities.

- Non-historised **satellite tables** recording the attributes associated with the hub or link table.

As a reminder here are the data vault tables

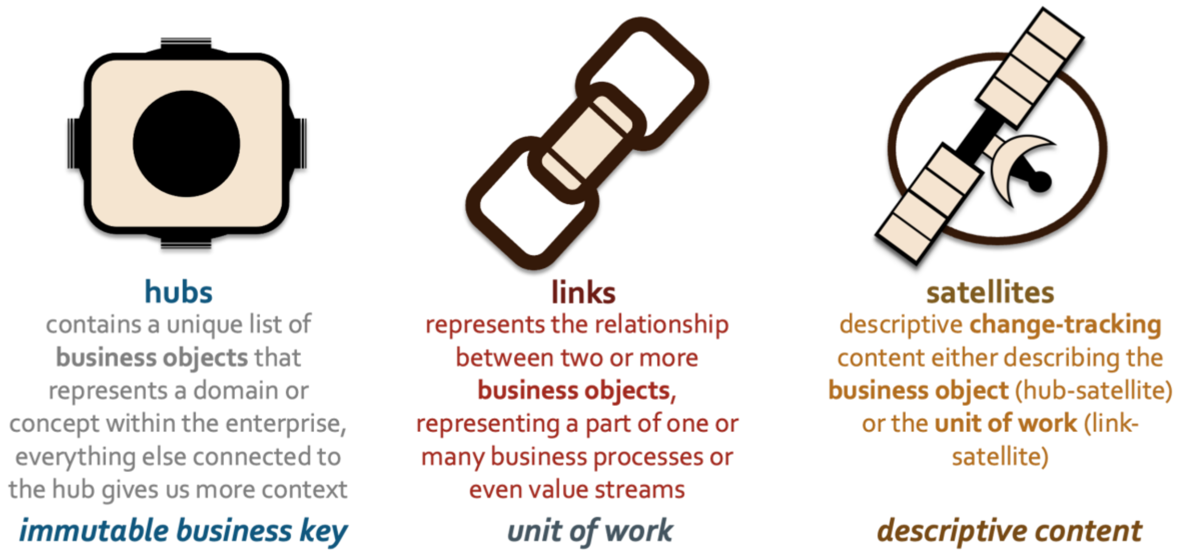

hubs
contains a unique list of business objects that represents a domain or concept within the enterprise, everything else connected to the hub gives us more context
immutable business key
links
represents the relationship between two or more business objects, representing a part of one or many business processes or even value streams
unit of work
satellites
descriptive change-tracking content either describing the business object (hub-satellite) or the unit of work (link-satellite)
descriptive content

Figure 3 Hubs, Links and Satellites

We previously discussed the recommended satellite table structure for semi-structured data in this article, bit.ly/3XNGFUU, the data needed for an activity stream will be loaded into raw vault. The type of satellite table you model this into depends on:

- If the data arrives as a batch file, the content should be modelled into a raw vault **satellite table**.
- if the data arrives near real-time (NRT) the content should be modelled into a raw vault **non-historised satellite** or **link table**.

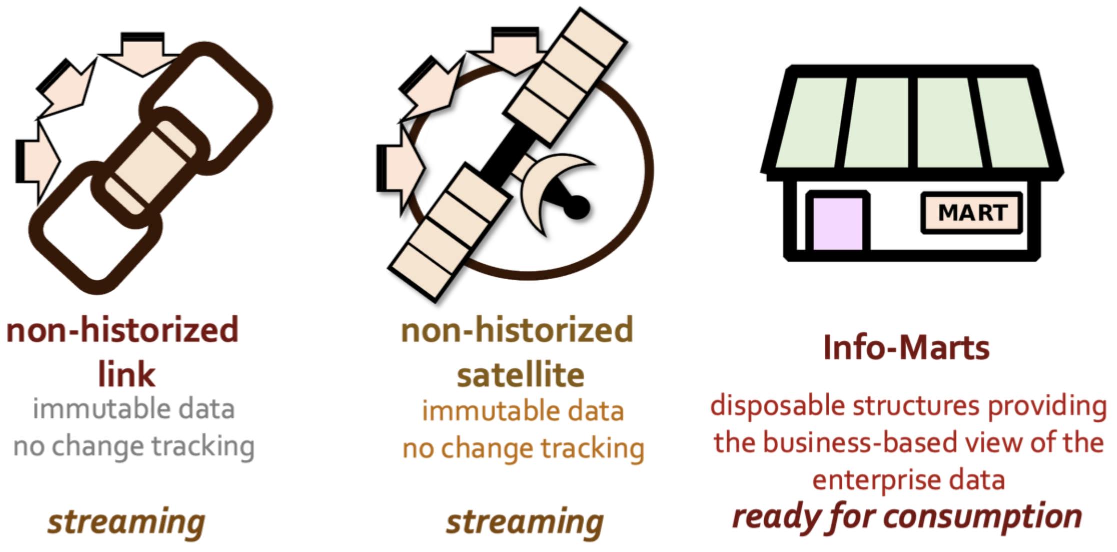


Figure 4 Non-historised Links and Satellites, Information Marts

With the payload modelled as raw vault artefacts the transformation of this data into an activity schema conforms that data for further analytics and applies structural and transformational business rules on that raw data. Therefore, activity schema is a **business vault** artefact modelled as a satellite table. For the modelled activities the cardinality should be one-to-one with the one or many raw data payloads from where the data came from and

persisted as an auditable source just like the raw vault payload is; it will contain the same metadata and applied date-timestamp as raw data but only contain the *business* activities we need. The applied date timestamp will match but the load date timestamp will differ between raw and business vault.
Temporal joins and other activity enrichment is performed in the information mart layer of your data vault architecture; hold that thought, let's demonstrate how we will apply all these concepts using Snowflake!

Activity Schema + Data Vault on Snowflake

Snowflake's capabilities continue to scale as rapidly as cloud technology scales for data. Today we can define an efficient application of the above framework at scale, and should your business use case require it, the above framework can process your data with less than a minute latency. Here's how,

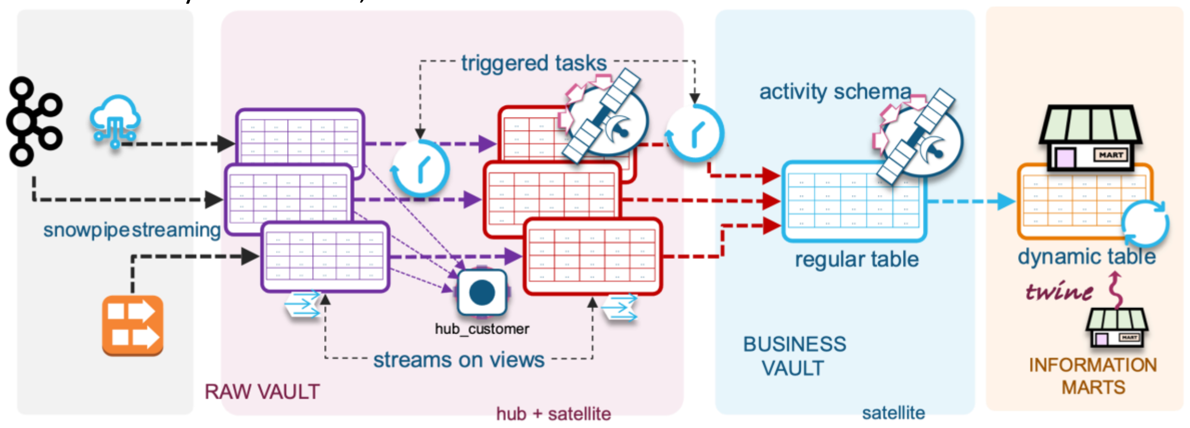


Figure 5 Continuous flow within Snowflake

Data ingestion – NRT or Batch

Snowflake includes the capability to process batch, micro-batch and near real-time data in the form of:

- COPY INTO that essentially converts supported table formats into Snowflake proprietary tables. The files must be available for ingestion from a staged area whether that's external (your cloud storage bucket) or internal (Snowflake's cloud storage bucket). We have a large network of partners who essentially wrap the COPY INTO statement into their tooling to get your data into Snowflake with the benefits those tools bring to your analytics platform.
- Snowpipe for micro-batched files is essentially Snowflake's own wrapper for the COPY INTO with some limitations.
- Snowpipe Streaming skips staging the data as files and loads your data directly into Snowflake tables as rowsets. The typical ingestion source for Snowpipe Streaming is Kafka which Snowflake has a supported connector.

COPY INTO and Snowpipe will not provide less than a minute latency however Snowpipe Streaming will. Additionally, event data may be made available as External Tables (with supported file formats), Iceberg Tables and Snowflake Data Sharing (this data is real-time). Data is now in Snowflake as raw content, onto the next step…

Model to Raw Vault – single or multiple topics

Typically for NRT data you need the data available to your business users as quickly as possible, therefore it stands to reason that **no** hashing is performed. That's right, for NRT data the act of applying hashing to:

- Define the surrogate hash key is costly and unnecessary and not recommended, we tested the effect of hashing data in Snowflake here.
- Since your target raw vault tables are non-historised (event data is new by definition), checking that the record you're ingesting is a **true change** is not necessary. Therefore, there is no need to build a hash-based record digest to check if the record you're ingesting is new (no HashDiff column). The reason why this is not necessary is because Kafka + Snowpipe Streaming guarantees exactly once semantics.

Processing of raw data into raw vault modelled data will use streams on views applied on the raw data. The act of adding data vault metadata tags to ingested data is thus repeatable and processing that stream on view ensures a consistent application of those metadata columns. Data will be processed as soon as there is new data in that stream when coupled with Snowflake's Triggered Tasks. If you chose to add surrogate hash keys to your event data then this is where they are defined.

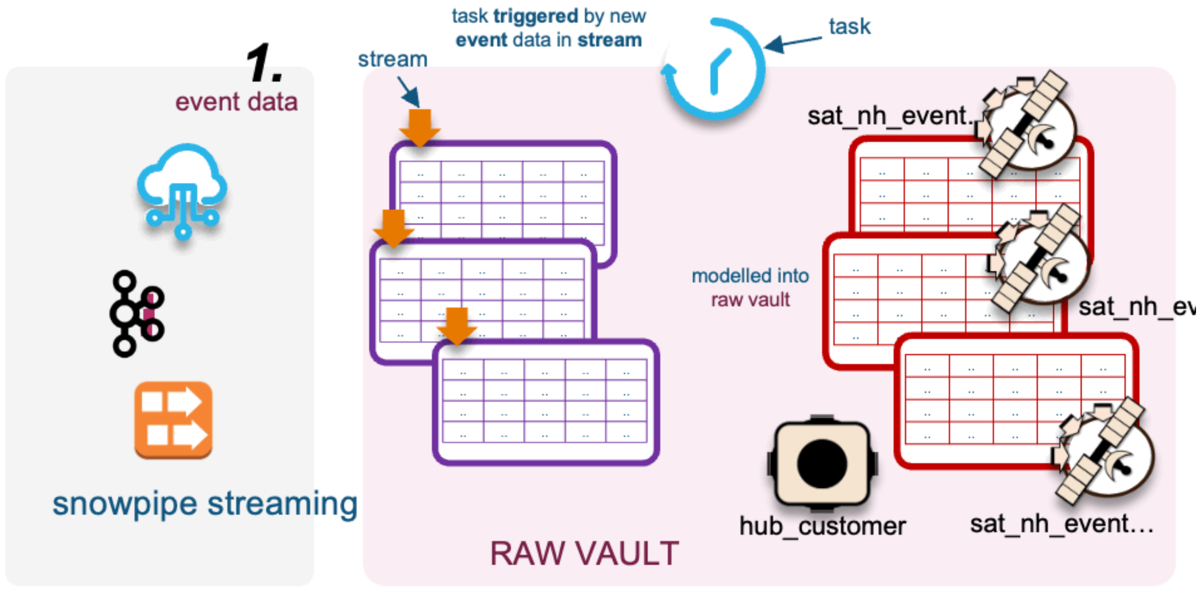


Figure 6 Raw data is streamed; Triggered Task detects new data to process into Raw Vault

```
create or replace task
staged.tsk_xero_page_activity_to_hub_customer_dv_hashkey_hub_customer
warehouse = vault
when
system$stream_has_data('str_xero_page_activity_to_hub_customer_dv_hashkey_hub_c
ustomer')
as
insert into datavault.hub_customer
from staged.str_xero_page_activity_to_hub_customer_dv_hashkey_hub_customer;
```

```
alter task staged.tsk_xero_page_activity_to_hub_customer_dv_hashkey_hub_customer
resume;
```

Recall the recommended framework for semi-structured raw vault satellite tables we published here. We will use that same framework for our non-historised raw vault satellite. Optionally, you can have schematization enabled for your Snowpipe Streaming ingestion and therefore you should have those same schematised columns in your raw vault satellite table. With a Stream on View deployed on your streaming data we will use Snowflake Triggered Tasks to grab new data to insert into the raw vault satellite table. This is similar framework we discussed in this blog on the Kappa Vault framework.

Model to Business Vault as an Activity Schema

We may have several raw vault satellites based on event data we may want to use in our activity schema. The processing of new raw data into activities would thus be performed by simply executing Snowflake's Triggered Tasks on Streams on Raw Vault Views to get that data from one or multiple sources into a single business vault activity schema. Locking the business vault satellite table is not an issue because each raw vault satellite will not have competing events and as we discussed earlier, every event is new by definition, we will not end up with duplicate activity data. We will persist with the standard activity schema columns we described earlier and use Snowflake's native support for semi-structure SQL operators and functions on the raw vault data to get the business vault satellite construct we want. And here is the construct, we already support some of the columns Activity Schema says is mandatory thus we will simply reuse those data vault columns here.

Column	Description	Data Type	Required
activity_id	Event_id	Text	Yes
ts - dv_applied	UTC timestamp; when the event occurred	Timestamp	Yes
customer (the business key)	Global identifier for the business entity	Text	Yes
activity	Formatted as a *verb_noun* the event is conformed into this activity we do analytics on. Examples include 'viewed_page', 'open_support_ticket'	Text	Yes
anonymous_customer_id	Local identifier, surrogate key used in the event system	Text	No
feature_json	Contains the activity-based content of interest, the crux of what makes each activity different	Variant / object	Yes

Column	Description	Data Type	Required
revenue_impact	Monetary value of the event (if any)	Numeric	Yes
link	Link to the web or mobile page where the event occurred	Text	Yes

Activity Schema includes two optional columns in the specification; however, these columns require an SQL update to the activity schema table. Data Vault 2.0 deprecated SQL UPDATES due to the operation being expensive, they have another operation that is not desirable for event data, event data is immutable and if we have millions of events the SQL UPDATE operation will be even more costly.
Thus, these update columns are reserved for the Information Mart portion of the implementation.

Column	Description	Data Type	Required
activity_occurence	Computed after; the number of times a given entity has performed the activity at the time the current activity has occurred.	Numeric	No
activity_repeated_at	Computed after; timestamp of the next time the activity is repeated	Timestamp	No

To conform the raw event data into an Activity Schema, repeatable transformation rules are applied using SQL that is explicit on the content selected from the raw data vault.

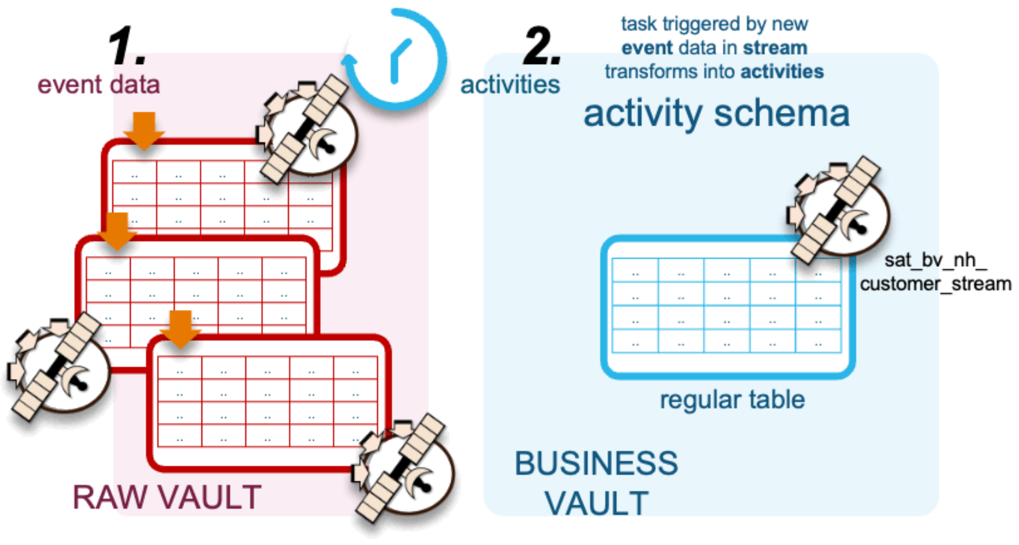

1.
event data
RAW VAULT
2.
activities
task triggered by new event data in stream transforms into activities
activity schema
sat_bv_nh_customer_stream
regular table
BUSINESS VAULT

Figure 7 Activity Schema is streamed; Triggered Task detects new data to process into Business Vault

Selecting activities can be as simple as running queries like this

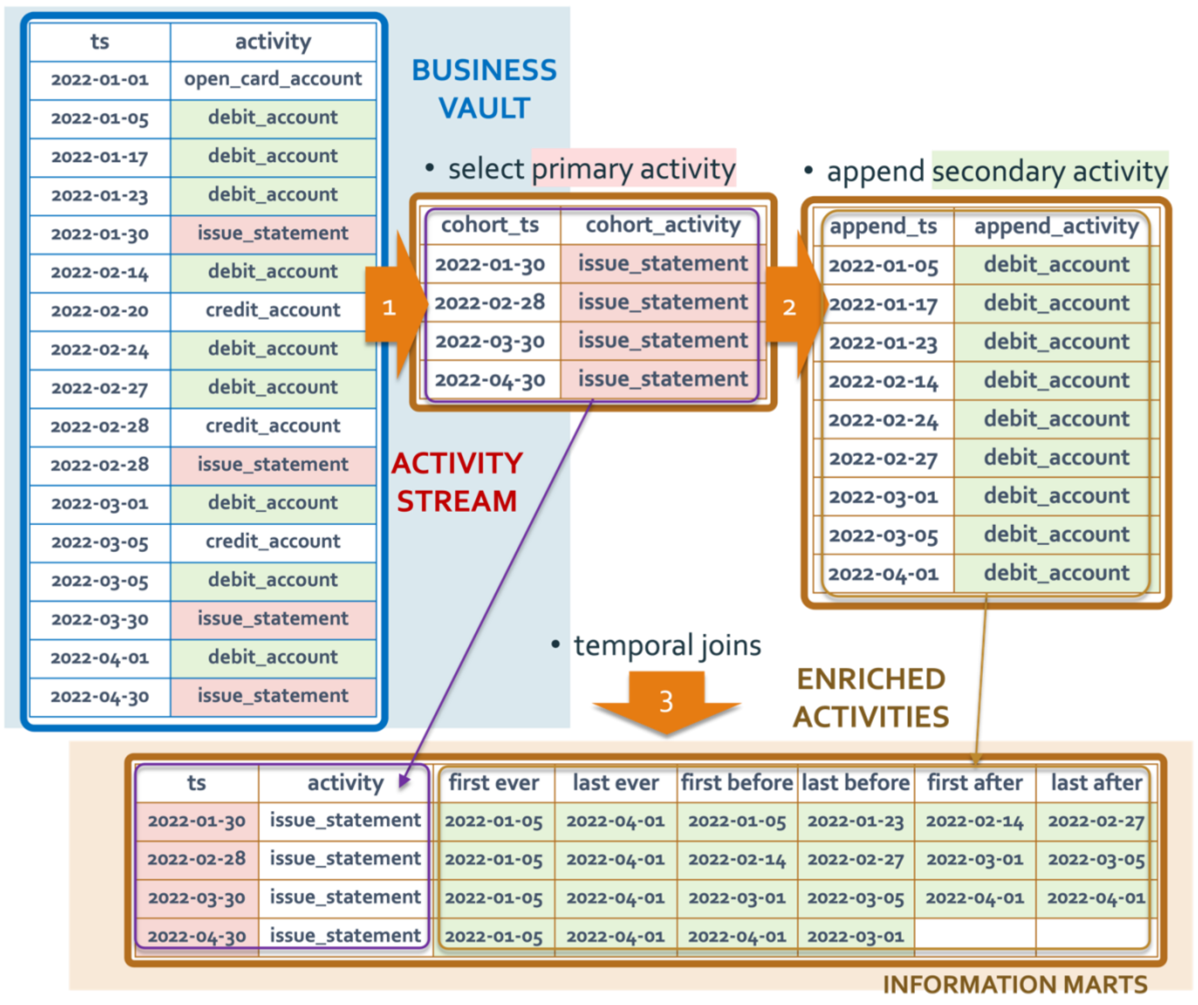
```
select
  parse_json(s.dv_object):"custcode"::text as customer_id
, parse_json(s.dv_object):"event-id"::text as activity_id
, to_timestamp(parse_json(s.dv_object):"timestamp"::text) as ts
, parse_json(s.dv_object):"details"::variant as feature_json
, case when parse_json(s.dv_object):"event"::text = '10' then 'issue_statement'
         when parse_json(s.dv_object):"event"::text = '2' then 'debit_account'
         when parse_json(s.dv_object):"event"::text = '3' then 'credit_account'
         when parse_json(s.dv_object):"event"::text = '1' then 'open_account'
   end as activity
, case when parse_json(s.dv_object):"event"::text in ('2', '3')
     then parse_json(s.dv_object):"details":amount::float
     else 0 end as revenue_impact
from payload s
where parse_json(s.dv_object):"event"::text in ('10', '2', '3', '1')
;
```

Enrich the Activity Schema with Self-Joins

Finally, to transform our data into a form enriched and palatable for our business analytics requirements; we deploy our information marts either as SQL views or as Snowflake Dynamic Tables.

The process behind Activity Schema's twelve templated temporal joins starts by selecting a primary activity (the cohort activity) and then appending (adding columns to the right of the cohort) with the following occurrences of the secondary activity:

- **First ever** – the first ever secondary activity
- **First before** – the first secondary activity that occurred before the cohort activity
- **First after** – the first secondary activity that occurred after the cohort activity
- **First in between** - the first secondary activity that occurred before the cohort activity and the next cohort activity
- **Last ever** – the last ever secondary activity
- **Last before** – the last secondary activity that occurred before the cohort activity
- **Last after** – the last secondary activity that occurred after the cohort activity
- **Last in between** - the last secondary activity that occurred before the cohort activity and the next cohort activity
- **Aggregate all ever** – aggregation of all secondary activities
- **Aggregate all before** - aggregation of all secondary activities before the cohort activity
- **Aggregate all after** - aggregation of all secondary activities after the cohort activity
- **Aggregate in between** - aggregation of all secondary activities that occurred before the cohort activity and the next cohort activity



ts	activity
2022-01-01	open_card_account
2022-01-05	debit_account
2022-01-17	debit_account
2022-01-23	debit_account
2022-01-30	issue_statement
2022-02-14	debit_account
2022-02-20	credit_account
2022-02-24	debit_account
2022-02-27	debit_account
2022-02-28	credit_account
2022-02-28	issue_statement
2022-03-01	debit_account
2022-03-05	credit_account
2022-03-05	debit_account
2022-03-30	issue_statement
2022-04-01	debit_account
2022-04-30	issue_statement

cohort_ts	cohort_activity
2022-01-30	issue_statement
2022-02-28	issue_statement
2022-03-30	issue_statement
2022-04-30	issue_statement

append_ts	append_activity
2022-01-05	debit_account
2022-01-17	debit_account
2022-01-23	debit_account
2022-02-14	debit_account
2022-02-24	debit_account
2022-02-27	debit_account
2022-03-01	debit_account
2022-03-05	debit_account
2022-04-01	debit_account

ts	activity	first ever	last ever	first before	last before	first after	last after
2022-01-30	issue_statement	2022-01-05	2022-04-01	2022-01-05	2022-01-23	2022-02-14	2022-02-27
2022-02-28	issue_statement	2022-01-05	2022-04-01	2022-02-14	2022-02-27	2022-03-01	2022-03-05
2022-03-30	issue_statement	2022-01-05	2022-04-01	2022-03-01	2022-03-05	2022-04-01	2022-04-01
2022-04-30	issue_statement	2022-01-05	2022-04-01	2022-04-01	2022-03-01		

Figure 8 Templated Cohort enrichment

Through combining cohort and append activities we can borrow features between cohorts and secondary activities. We may even decide to append the same activities to the cohort activity.
Imagine if the cohort record appended with surrogate sequence id from your satellite table instead of activities powering your SNOPIT… very powerful indeed!

How did we do it?
As of today, Dynamic Tables are not yet fully matured and thus to keep this implementation as efficient as possible for the time been we have split the enrichment into three dynamic tables (ideally, we would want this implemented as a single Dynamic Table). The reason we do this is simple, the goal of Dynamic Tables is to process the transformations in the select statement *incrementally*, unfortunately some transformations still require the Dynamic Table to perform a full refresh at every update.

Change tracking is **not** supported on queries with window functions that have disjoint partition keys.

Given time these limitations will reduce, and this implementation will be implemented as a single Dynamic Table instead.

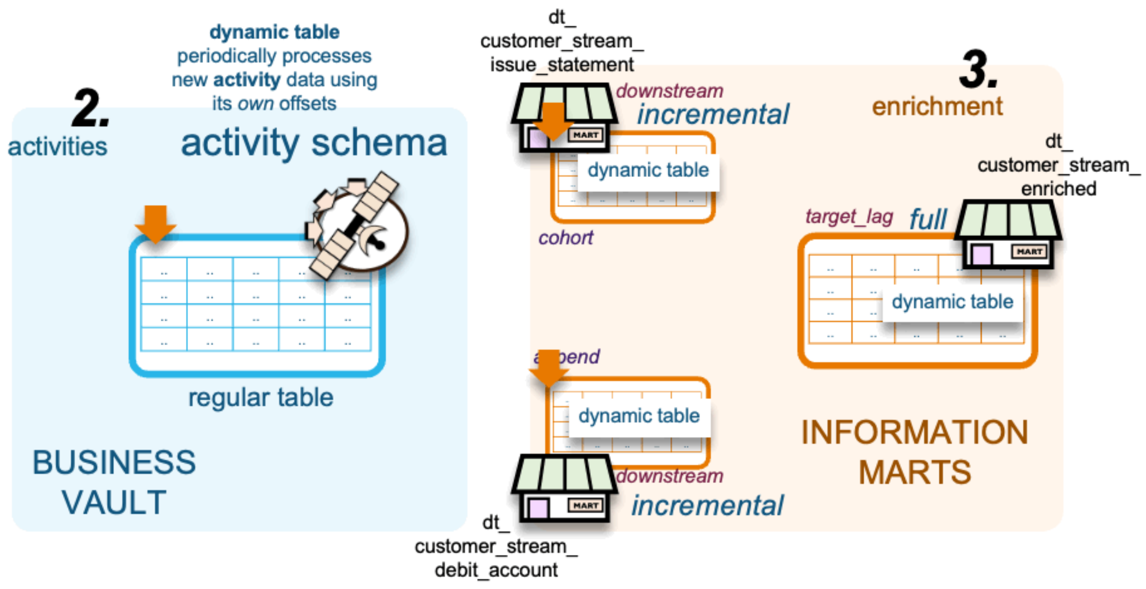
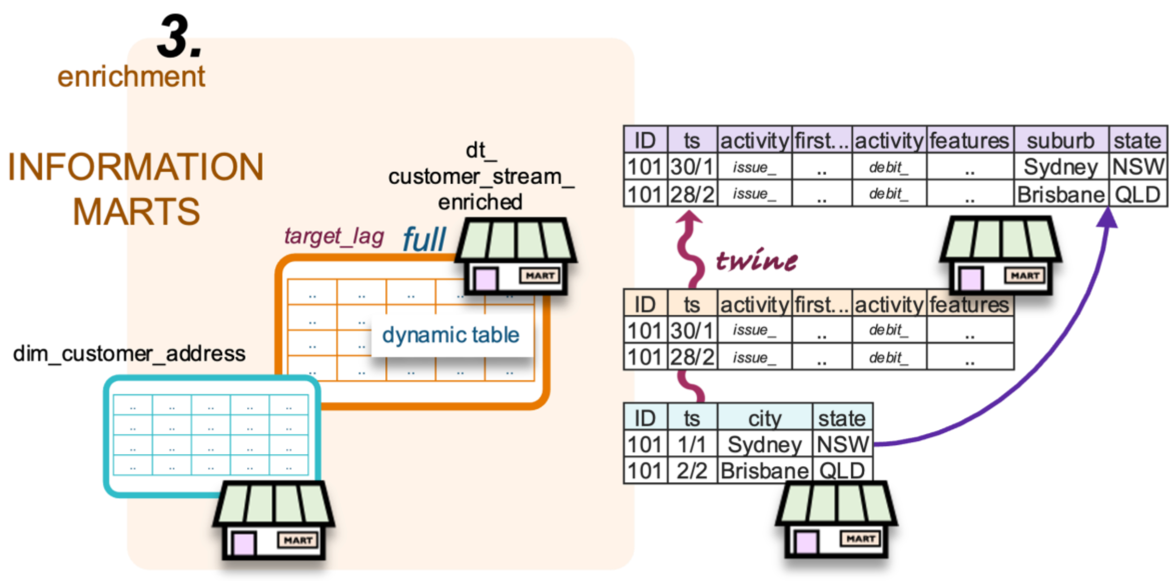


Figure 9 Activity Schema is enriched as Dynamic Tables

Enhance with Twine

Combining dimensional and time-series data is a little different to joining Kimball facts and dimensions. Because time-series data is meant to be processed in near real-time a more efficient join is to enrich time-series data *on the fly* and in Snowflake that is efficiently processed as an as-of join instead of a range join. As new time series data arrives by using the as-of join the applicable point-in-time dimensional record is *picked*. Asof joins can be used to combine multiple stream data together, but it is perfectly fine to use the same construct to enrich activity data with dimensional data.



ID	ts	activity	first...	activity	features	suburb	state
101	30/1	issue_	..	debit_	..	Sydney	NSW
101	28/2	issue_	..	debit_	..	Brisbane	QLD

ID	ts	activity	first...	activity	features
101	30/1	issue_	..	debit_	..
101	28/2	issue_	..	debit_	..

ID	ts	city	state
101	1/1	Sydney	NSW
101	2/2	Brisbane	QLD

Figure 10 Simplified twine, as of JOIN

```
select a_s.customer_id, cohort_ts
       , line1, line2, suburb, pcode
```

```
        , cohort_activity_repeated_at
        , first_ever_ts, last_ever_ts, first_before_ts, last_before_ts
        , first_after_ts, last_after_ts
        , a_s.cohort_feature_json
        , before_aggregate_count, revenue_impact_before
        , after_aggregate_count, revenue_impact_after
-- borrowed features
        , first_before_feature_json, last_before_feature_json
        , first_after_feature_json, last_after_feature_json
from information_marts.dt_customer_stream_enriched a_s
asof join information_marts.dim_customer dim
match_condition(a_s.cohort_ts >= dim.dv_applieddate)
on a_s.customer_id = dim.customer;
```

If the underlying enriched activity schema is a full refresh so too will the dynamic table that is using the asof join be. Hopefully it won't be too long before dynamic tables support more SQL semantics to enable complete incremental loads for all your streaming data!

Input Dynamic Tables are FULL refresh: [DT_CUSTOMER_STREAM_ENRICHED]

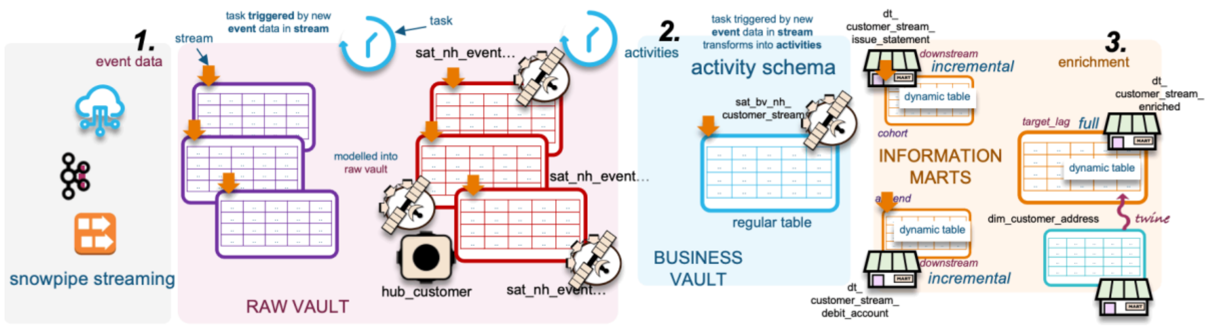
And that completes the repeatable event data pattern using a combination of activity schema, data vault and Snowflake.

In Summary

A design pattern of data mesh architecture is using EDA to *inform* all other domains of that domain's events as topic subscribers, *asynchronously*. Thus, an efficient, repeatable, and auditable pattern for your analytics based on event data using data vault patterns can leverage Snowflake's own ever-growing list of capabilities, including

- Streams on Views to deploy offsets on underlying tables supporting that view
- Triggered Tasks that executes the associated transformation based on new data
- Dynamic Tables that execute declarative SQL to process new data and is self-healing and incremental.

The work for you is to model the activities you need and because Snowflake is a Software-as-a-Service (SaaS); Snowflake will manage the rest! Once you're set up, "turn on the tap", i.e. when the data from your streaming source starts to flow in, nothing else in your data pipeline requires scheduling or any orchestration. Triggered Task picks up the new raw data, triggered task picks up the new raw vault data and dynamic tables pick up the new business vault data.



As you may have noticed, in an Activity Schema we pick the cohort activity, append the secondary activity, and use the repeatable temporal join patterns to enrich that data. That means *any* of the modelled activities could be selected as the cohort activity and other activities (including the same activity) can be appended, yes, Activity Schema from start to finish is a set of repeatable patterns. This is why activity schema is so attractive to data vault, the only difference between picking an activity is controlled by setting a parameter in your automation,

- cohort_activity = issue_statement; append_activity = debit_account
- cohort_activity = debit_account; append_activity = issue_statement
- cohort_activity = issue_statement; append_activity = issue_statement
- cohort_activity = debit_account; append_activity = debit_account

Or any other modelled activity you have designed.

A common question I get asked when it comes to streaming data is whether that data should skip the data vault, I have just shown you that it doesn't! Data vault remains your integrated repository for auditable data and using Snowflake your analytics will be provided in near real-time.

There was obviously a lot of usage of the word *stream* in this article, it's unfortunate but to clarify let's break down its usage here:

- **Activity Stream** – timeseries data modelled for analytical use cases. Not related to any technology.
- Snowflake Streams – data object within Snowflake that applies an offset to underlying data object it is created on. At the time of writing the supported objects you can add an offset to are:
    - Snowflake tables
    - Snowflake views (applies the offset to the underlying tables)
    - External tables registered in Snowflake
    - Snowflake shares
    - Directory tables

  This is completely within Snowflake.
- Snowpipe Streaming – Snowflake support for NRT data using an event stream, a common source is Kafka. Processing new data as a subscriber also uses an offset but this is on the Kafka topic itself.

All three are asynchronous; I hope that clarifies everything for the reader!

References

- Designing Event-Driven Systems (Ben Stopford), bit.ly/3ujq2WN

- Designing Data-Intensive Applications (Martin Kleppmann), bit.ly/47GJXNZ
- Turning the database inside-out (Martin Kleppmann), bit.ly/49FoGWH
- Activity Schema 2.0 Spec (Ahmed Elsamadisi. narratordata.com), bit.ly/47lW3Ms
- Activity Schema Temporal Joins, bit.ly/49PvQHL
- Temporal Dimensional Modelling (Lars Rönnbäck, twine), bit.ly/3GdSz2O
- Kappa Vault, bit.ly/3JbRf05

Note that **asof joins** and **triggered tasks** are in Private Preview at the time of writing. The syntax might change when these features move into Public Preview or General Availability

The views expressed in this article are that of my own, you should test implementation performance before committing to this implementation. The author provides no guarantees in this regard.