Persisted Staging Entity Design Reference

Table of Contents

Authors

James Moran 1/6/2026

Introduction

What follows is the design reference for persisted staging modeled entities in a warehouse environment. As noted in the below terminology and definition section, a persisted staging area is optional area in the data solution design that records the transactions (events) that were received by the data solution over time. A Persistent Staging Area (PSA) can be considered a type of data warehouse insurance policy where all data is captured and has the following attributes.

- Source system data is loaded into PSA without transformation (with minimal data type transformation--where there are no equivalent target data types for source data types)
- Records are never deleted from PSA (archives may occur)
- PSA stores all unique records loaded (tracks history)
- Many more fields are stored in PSA than required by the data warehouse

The end goal here is to use this PSA to capture all the data that the data warehouse receives to ready it to be integrated in an integration data layer, and after integration load it into a dimensional modeled enterprise data warehouse layer to be used in OLAP based reporting. One

of the benefits of this approach is that with all the data captured in persisted staging, the integration and enterprise data warehouse layers can be fully re-reloaded in the event of structural changes to the tables. This is something that is difficult using any of the older data warehouse staging approaches because of the temporal nature of the data.

Other Staging Approaches

Persisted staging is a newer method that is now possible because the cost of disk storage space has come down significantly. In the past, in classic Inmon data warehouse thinking, there was no persisted staging due to the cost of RDMS disk space and the added overhead on the older scale up nature of the RDBMS, so therefore data was integrated directly from non-persisted staging, usually OS disk files, into a 3rd normal form modeled data warehouse environment by the Integration and Transformation layer which was "made up primarily of programs". The 3rd normal form structures in the resulting data warehouse environment were a combined composite model of the source system structures; that is, like tables were assimilated into a single table structure. This method was later amended and the concept of a data vault that was modeled using a hub, link, and satellite entity structure was introduced by Dan Linstedt. Data vault is a sort of hybrid approach of the persisted staging approach discussed here and the Inmon approach as the satellites for each hub are modeled closely to the source system structures in what is called the raw vault, with the final assimilated single table satellites created in the business vault. This approach was possible because of the lowering of disk space costs and some design efficiencies having to do with data decomposition inherent in the approach. Both of these approaches can be contrasted to Classic Kimball thinking which was to directly load the dimensional modeled data warehouse tables from the non-persisted OS disk files using intermediate de-normalized staging structures. This was usually found to be a much faster development approach but this also made schema change much more difficult because of the temporal nature of the dimension tables. Interestingly all of the approaches do not handle schema change well, but the Kimball approach is considered the hardest to manage.

Persisted Staging

As noted, the older stages approaches were used primarily during a time where disk space was expensive (in the case of the data vault approach disk space cost was coming down) and RDBMS space was at a premium due to the scale up nature of the RDBMS's. Currently, space is inexpensive, loads are very fast using scale out technology, and overall storage, when the underlying RDBMS storage is columnar, is much more efficient than RDBMS row storage. Given this, you can store large quantities of raw data in a RDBMS in a cost effective manner. This has lead to a new concept called a persisted staging.

"What is a Persistent Staging Area?
A PSA is a record, or archive, of all data deltas (records) that were ever presented to the Data Warehouse (Staging Area) by its interfaces: historised raw data. Every record that is loaded into the Staging Area – a transient area (will be truncated regularly) – is sent to the PSA as well as to the upstream (usually Data Vault based) Integration Layer." [Voss 20190513]

In a way, the persisted staging method suggested looks a lot like Inmon 3rd normal form, and a bit like data vault modeled tables, but you dispense with up front modeling chores and instead create tables that are essentially exact copies of the source tables with some added metadata columns. In this version of the approach the author is suggesting, four tables are created for each one table in the source. A landing table that is simply a non-persisted 1st landing area for the data. A current table that is merged into to enable incremental loads that have better performance, and for agile report development on current valued volatile data. A historical table that is used to track all changes that come into the data warehouse and is similar to the type 2 dimension tables that are instantiated in a Kimball dimensional model. And finally, a current key snapshot table that captures deletes on source systems by capturing only the current source key sets at on the periodic load time so the key set can be compared to the current table and if the key is not in the current key snapshot table it can be soft deleted in the current table with a deleted row added to the history table.

As noted; the structure of these persisted stage tables, the current table and historical table, will resemble the hub to satellite structures found in data vault modeling with the current table acting as a hub table and the historical table acting as a satellite table, but because the structure is source oriented the method eases the modeling work (it can be automated in fact).

Summary

In the end, a developer still has to design the integration model and code to integrate the various sources into a Kimball dimensional model, but with persisted staging the costly time consuming design of a raw vault can be avoided.

Additional Notes

Constant case is used throughout this document both in the syntax conventions and syntax examples.

1. Objectives

1.1. Data Governance

1.1.1. Readability and Consistency

The goal of this naming convention is to aid in readability and consistency.

1.1.2. Data Type and Size Standardization

Data types for the same attributes should agree across name-spaces to aid in query performance, comparisons, joins, etc. A goal of this standard is to begin normalizing those sizes and types.

1.1.2.1. Base2 Varchar Size Consistency

A size that is a product of 2^x is recommended to arrive at a VARCHAR size that is bigger than a source. This builds in some extra space for potential column size expansion in source systems and aids in data type size consistency. Although all VARCHAR sizes *could be* defined as the max allowed value with only the size used being stored, that is not usually considered optimal given that some BI/ET tools define the maximum size of the VARCHAR data in storage or in memory.

2. Terminology and Definitions

2.1. General Terms

2.1.1. Persisted Staging Area

A Persistent Staging Area (PSA) is an optional area in the data solution design that records the transactions (events) that were received by the data solution over time. Much like an archive, it shows the changes in data in its original form and prior to any interpretation towards data integration or delivery.

2.1.2. Dimensional Modeling

"Dimensional modeling (**DM**) is part of the *Business Dimensional Lifecycle* methodology developed by Ralph Kimball which includes a set of methods, techniques and concepts for use in data warehouse design. The approach focuses on identifying the key business processes within a business and modelling and implementing these first before adding additional business processes, as a bottom-up approach." [Wikipedia 1 n.d.]

2.1.3. Database Normalization

"Database normalization or **database normalization** (see spelling differences) is the process of structuring a relational database in accordance with a series of so-called ***normal forms*** in order to reduce data redundancy and improve data integrity. It was first proposed by British computer scientist Edgar F. Codd as part of his relational model." [Wikipedia 2 n.d.]

2.1.4. Database Denormalization

"Denormalization is a strategy used on a previously-normalized database to increase performance. In computing, denormalization is the process of trying to improve the read performance of a database, at the expense of losing some write performance, by adding redundant copies of data or by grouping data.[1][2] It is often motivated by performance or scalability in relational database software needing to carry out very large numbers of read operations. Denormalization differs from the unnormalized form in that denormalization benefits can only be fully realized on a data model that is otherwise normalized." [Wikipedia 3 n.d.]

2.1.5. Name Normalization

“The task of name normalization, as it is defined, involves the mapping of a phrase/word to a unique concept in an ontology (based on the description of that concept in the ontology) after disambiguating potential ambiguous surface words, or phrases, and then assigning the concept’s unique identifier in the ontology to the name.” [D’Souza 2015]

2.1.6. Schema Evolution

“Schema evolution is the process of modifying the structure of a database over time as new requirements and data models emerge. It involves changing the schema of tables to accommodate new data elements or changes to existing ones.”[IBM 20250711]

E.g. adding a new column to a table is an example of evolving the schema.

2.2. Syntax Conventions

Microsoft Transact SQL Syntax conventions are used for clarity in this document when naming conventions and commands are documented.

Convention	Used for
UPPERCASE	Transact-SQL keywords.
italic	User-supplied parameters of Transact-SQL syntax.
bold	Type database names, table names, column names, index names, stored procedures, utilities, data type names, and text exactly as shown.
underline	Indicates the default value applied when the clause that contains the underlined value is omitted from the statement.
| (vertical bar)	Separates syntax items enclosed in brackets or braces. You can use only one of the items.
[ ] (brackets)	Optional syntax items. Don't type the brackets.
{ } (braces)	Required syntax items. Don't type the braces.
[,...n]	Indicates the preceding item can be repeated n number of times. The occurrences are separated by commas.
[...n]	Indicates the preceding item can be repeated n number of times. The occurrences are separated by blanks.
;	Transact-SQL statement terminator. Although the semicolon isn't required for most statements in this version of SQL Server, it will be required in a future version.

<label> ::=	The name for a block of syntax. Use this convention to group and label sections of lengthy syntax or a unit of syntax that you can use in more than one location within a statement. Each location in which the block of syntax could be used is indicated with the label enclosed in chevrons: <label>.

[MS Learn 1 n.d.]

3. Persisted Staging Entity and Attribute Structure

3.1. Persisted Staging Architecture Discussion

Native Persisted staging is comprised of four table types; Landing, current, historical, and in cases where deletes cannot be handled by any other means, a current key snapshot.

3.1.1. Landing

Landing tables are constraint free landing entities that are essentially the first stop for data when it is copied into the database environment. Any data type conversion issue will be found here when data is copied into these landing tables.

3.1.2. Current (Inmon ODS style)

The second stop is the current stage table. These tables are essentially structural copies of source tables that are persisted in a Inmon ODS like manner; subject-oriented, current valued, volatile, and detailed, but not integrated. Data is first merged into these current tables to optimize raw data loading (the current tables will eventually be a lot smaller than the historical tables so merging data is more efficient), to aid current raw data reporting, and to optimize the historical loads.

3.1.3. Historical (Inmon EDW style)

The final stop in the persisted staging area are the historical tables. These tables are also essentially structural copies of the source tables, but are persisted in a Inmon data warehouse like manner; subject-oriented, time-variant, nonvolatile, but not integrated.

3.1.4. Current Key Snapshot

The current key snapshot table is used to load a narrow version of the entire source row set for post load soft delete processing. Essentially, when a key is not in this table, the data in the current table is soft deleted (the __PSTAGE_DELETED_INDICATOR indicator is set to True). The processing will flow down into the historical table when this occurs.

3.2.1. Database Normalization Not Required

The structure of the entities (tables) in the persisted staging area should follow the structure of the source tables or feeds including any denormalization of those feeds by the providers. The primary goal with the persisted staging area is to preserve the data in its raw state and to make it operable upon by extract programs, views and end user query tools.

3.2.2. Persisted Entity Naming\Renaming

Source entity names may need to be slightly modified for use within the target RDMBS when entities are created based on those sources in the persisted staging area to aid in readability and consistency. Source tables that have names that collide with reserved words in the RDBMS will need to be renamed when an entity is created as a table in the RDBMS to avoid the use of quotes in code. In addition, the source table names will need to have the proper suffixes attached to identify their purpose and to not collide with like named tables in the same schema. Below are guidelines on that process.

3.2.3. Source Table Name Name Normalized with Persisted Staging Suffixes

3.2.3.1. Name Normalization

The source entity name will be name normalized using either lower snake case or constant case (term separation by underscores), with the following specifics:

Collisions with RDBMS SQL reserved words will necessitate an altering of the name. The developer should use their best judgment when doing this.

3.2.3.2. Persisted Staging Suffixes

For landing tables, that is tables that are simply used as holding areas for rows of tabular data prior to load into the staging tables, the source table name should be name normalized and a suffix of __LAND added. Note that the underscore in __LAND is a double underscore.

For current valued (ODS) tables (insert new records and update any existing records with changes) the source table name should be name normalized and one of two methods can be chosen, either a suffix of __CRNT is added, or the suffix can be left off entirely and the current table can be thought of as a main raw reporting table. Note that the underscore in __CRNT is a double underscore.

For history mode tracking tables tables (EDW tables that track history using a type 2 approach) the source table name should be name normalized and a suffix of __HIST added. Note that the underscore in __HIST is a double underscore.

For more specifics see the build template below.

3.3. Persisted Staging Build Template

Use the following template to build the entity names.

3.3.1. Landing Table Naming

{ *source table name converted to all caps snake case* }__LAND

3.3.2. Current Table Naming

Note the __CRNT is optional.

{ *source table name converted to all caps snake case* }[__CRNT]

3.3.3. Historical Table Naming

{ *source table name converted to all caps snake case* }__HIST

3.3.4. Current Key Snapshot Naming

{ *source table name converted to all caps snake case* }__CKS

3.3.5. Name Examples

CUSTOMER__LAND

CUSTOMER (optional suffix removed).

CUSTOMER__CRNT (with optional suffix).

CUSTOMER__HIST

CUSTOMER__CKS

CUSTOMER_HISTORY__LAND

CUSTOMER_HISTORY__CRNT

3.4. Persisted Attribute Naming\Renaming

Source attribute names may need to be slightly modified for use within the RDBMS when attributes are created based on external column sources to aid in readability and consistency. Source columns that have names that collide with reserved words in the RDBMS will need to be renamed when the attribute is created as a column in the RDBMS to avoid the use of quotes in code. Below are guidelines on that process.

3.4.1. Source Column Name Name Normalized to Constant Case

3.4.1.1. Name Normalization

The source column name will be name normalized using constant case (term separation by underscores), with the following specifics:

Collisions with the RDBMS SQL reserved words will necessitate an altering of the name. The developer should use their best judgement when doing this.

3.4.1.2. Name Examples

Some examples of names that follow the build process are below.

1. EMPLOYEE_ID
2. EMPLOYEE_BIRTH_DATE
3. DATE_KEY
4. GL_ACCOUNT_NUMBER
5. SALE_STATUS_CODE

3.4.2. Base2 Sizing for VARCHAR data types

All VARCHAR columns should be re-defined in powers of 2^x (2, 4, 8, 16, 32, 64, 128, 256, etc.) to aid consistency; that is, columns that exists in multiple name spaces can be normalized to a common length by diverse sets of developers working independently.

Note this should only be done for current and historical tables. Landing tables columns that are varchar should follow the source column sizing.

3.4.2.1 Base2 Sizing calculation

To arrive at source column lengths we calculate the target column by determining the factor of 2 to the power of x that is just greater than the source column size.

For example, if a source column length is 100 (and the class size given is too small) we would calculate the length for the current and historical table column like so.

1st try.

$2^6 = 64$

2nd try.

$2^7 = 128$

And would arrive at 128 as the field size because it is product of 2^x that is just above 100. The next size up $2^8 = 256$, would be considered in this case too large, because $2^7 = 128$ already satisfies the requirements.

Example python code is below that can do this calculation easily.

```
def base_2_length(input : int) -> int:
    return_value = 0
    base = 2
    exp = 1
    length = 0
    if input == 16777216:
        return_value = input
    else:
        while ((input - length) > 0):
            length = base**exp
            exp += 1
        return_value = length

    return return_value
```

3.5. Persisted Staging Table Standard Metadata

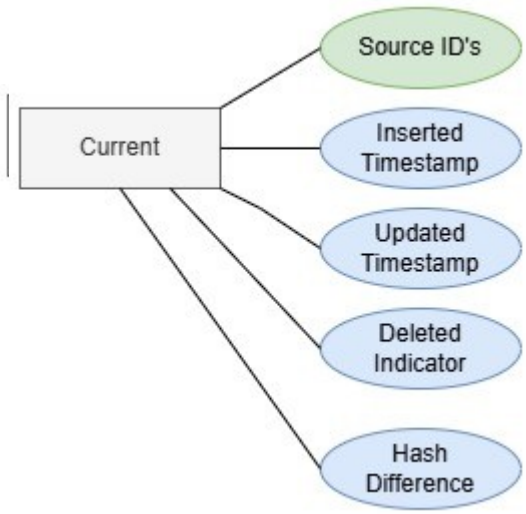
The persisted staging design has several standard metadata attributes.

3.5.1. Persisted Staging Metadata

The persisted stating metadata and primary key constraint information are noted below using Chen's database notation.

3.5.1.1. Landing Entity Persisted Staging Overview

Landing tables have only one metadata attribute column, the inserted timestamp. They do not have any primary key constraints set as they are essentially the initial entity landing data area. Any constraint checking will be done when merging the data into the current and historical tables.

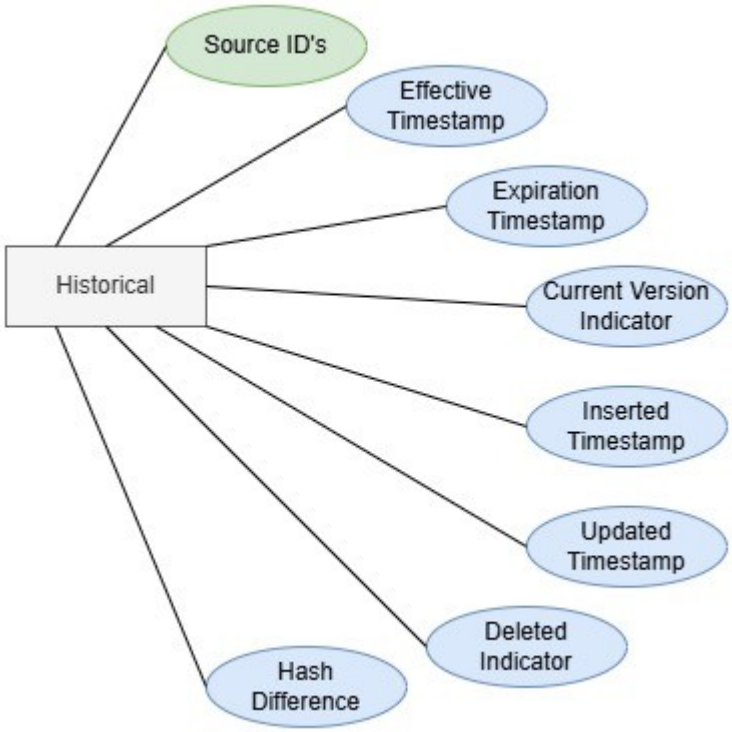


3.5.1.2. Current Entity Persisted Staging Overview

For current tables using a Chen's database notation you'll note the key in green and the standard non key metadata in light blue.

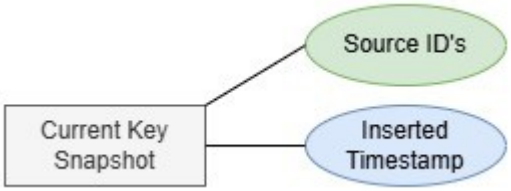
3.5.1.3. Historical Entity Persisted Staging Overview

For historical tables again using Chen's database notation you'll note the key(s) in green which now includes the native start metadata. Also note the non key metadata in light blue.



3.5.1.4. Current Key Snapshot Overview

For current key snapshot tables again using Chen's database notation you'll note the key(s). Also note the non key metadata in light blue. Note in these tables there are no other attributes to keep the table as narrow as possible.



3.5.2. Persisted Staging Attribute Metadata Details

3.5.2.1. Source ID or ID's.

Source Id's are the primary identifiers in source tables and can act as foreign keys in child tables.

Application

Current, Historical. and Current Key Snapshot tables.

Key Name Note

A primary key should be added to both the Current, Historical. and Current Key Snapshot tables for the key and named using the following template.

PK_{entity name}

Current Example:

```
...
CONSTRAINT PK_CUSTOMER PRIMARY KEY (CUSTOMER_ID)
);
```

Historical Example:

```
...
CONSTRAINT PK_CUSTOMER__HIST PRIMARY KEY (CUSTOMER_ID,
__PSTAGE_EFFECTIVE_TIMESTAMP)
);
```

Name Format

The ID name will follow the source ID's name exactly.

Data Type

Follows the source.

Nullability

Not null.

3.5.2.2. Persisted Stage Inserted Timestamp

Indicates the date and time the data was inserted in the current table, the date and time the data was inserted in a historical table, and the date and time the key was refreshed in the current key snapshot table.

Application

Landing, Current, Historical. and Current Key Snapshot tables.

Name Format

__PSTAGE_INSERTED_TIMESTAMP

Data Type

TIMESTAMP

Nullability

Null in landing tables, not null in current, historical, and current key snapshot tables.

3.5.2.3. Persisted Stage Updated Timestamp

Indicates the date and time the data was last updated in the current table and the date and time the data was last updated in a historical table. Note that in the historical table, this will usually indicate the date and time that the follow on update process has updated the needed time span metadata and current version indicator.

Application

Landing, Current, Historical. and Current Key Snapshot tables.

Name Format

__PSTAGE_INSERTED_TIMESTAMP

Data Type

TIMESTAMP

Nullability

Null in landing tables, not null in current, historical, and current key snapshot tables.

3.5.2.4. Persisted Stage Deleted Indicator

A true false indicator to indicate if the row has been deleted in the source system. Essentially this is a soft delete flag.

Application

Current and Historical Tables.

Name Format

__PSTAGE_DELETED_INDICATOR

Data Type

BOOLEAN

Nullability

Null.

3.5.2.5. Persisted Stage Hash Diff

This is MD5 hash of all of the columns to ease the change testing process when merging rows into current and historical tables.

Application

Current and Historical Tables.

Name Format

__PSTAGE_HASH_DIFF

Data Type

VARCHAR(32)

Nullability

Null.

3.5.2.6. Persisted Stage Effective and Expiration Timestamp

These attributes are in use for historical changes and represent the span of time that historical row is valid. Care needs to be taken in using these attributes when integrating different sources as these time spans are valid for the individual source only with other sources possibly superseding the time span for the same business key in an integration set. The Effective Timestamp indicates the beginning timestamp the row is valid for, and the Expiration Timestamp indicates the timestamp that the row valid to.

Application

Historical tables.

Name Format

__PSTAGE_EFFECTIVE_TIMESTAMP

__PSTAGE_EXPIRATION_TIMESTAMP

Data Type

TIMESTAMP

Nullability

Not null for __PSTAGE_EFFECTIVE_TIMESTAMP and null for __PSTAGE_EXPIRATION_TIMESTAMP.

3.5.2.7. Persisted Stage Current Version Indicator

A true false indicator to indicate if the row is the most current row in a historical table. Essentially this is the current row flag.

Application

Historical tables.

Name Format

__PSTAGE_CURRENT_VERSION

Data Type

BOOLEAN

Nullability

Null.

4. Persisted Staging Schema and Data Evolution

Over time, change will occur to tables in the source, and those changes will need to be implemented in the persisted staging entities. This section details how that is done.

4.1. General Notes.

4.1.1. Column Removal

Columns are not removed from persisted staging tables even if the columns are removed in the source. Although this is considered an anti-pattern (Blaha 8.2 Dead Elements Anti-pattern) in the case of persisted staging, it is a necessary exception in order to preserve the data as it was in the history tables. In the case of the removed column being part of the key, see the source key changes step below on how that is handled.

4.1.2. Schema Change verses Data Changes

This process does not track schema changes over time, and is not intended to. Industry experience has shown than the overhead of leaving table copies for each table change in place when the data has been moved into the new table structure has limited usefulness beyond the initial schema evolution, and leads to excessive table bloat. Table changes can be tracked separately by the code repository which is a more appropriate method if needed, but this is not recommended. With that said, there is a recommended amount of time to leave the temporary copies of the older tables in place.

Note: This process is designed to not indicate changes in data when the schema changes.

4.2. Source Key Changes

Source key changes require a complete renaming of the current and the historical tables to a deprecated namespace with a date and time suffixed so that the data in the state that it was in can be preserved. The landing table and the current table can be replaced after all loads have completed. Note the underscores in the naming template are double underscores.

4.2.1. Deprecated Current Table Naming

Note the __CRNT is optional.

{ *source table name converted to all caps snake case* }[__CRNT]__DEP__{ *date and time of deprecation* }

{ *date and time of deprecation* } = { *YYYYMMDDHHMM*[*AM* | *PM*] }

E.g. CUSTOMER__DEP__202410180318PM

4.2.2. Deprecated Historical Table Naming

{ *source table name converted to all caps snake case* }__HIST__DEP__{ *date and time of deprecation* }

{ *date and time of deprecation* } = { *YYYYMMDDHHMM*[*AM* | *PM*] }

E.g. CUSTOMER__HIST__DEP__202410180318PM

4.3. Change Steps for Column Addition

4.3.1. Landing Tables

Landing tables can be replaced in place after all previous loads have completed and the landing tables have been emptied.

4.3.2. Current Tables and Historical Tables

4.3.2.1. Changed Table Creation

The table with the added columns should be created using the following formats with the primary key named for the base persisted staging table with the schema evolution suffix with one caveat. If the base RDBMS allows for duplicate key names in the same schema, e.g. two tables in the same schema can have primary keys with the same name, then primary key renaming can be skipped.

4.3.2.1.1. Current Table Name

The table name will have a suffix indicator of MIG added (for migration) and the date and time of the table change and migration.

Note the __CRNT is optional.

{ *source table name converted to all caps snake case* }[__CRNT]__MIG__{ *date and time of table evolution* }

{ *date and time of table evolution* } = { *YYYYMMDDHHMM*[*AM* | *PM*] }

E.g. CUSTOMER (optional suffix removed) would be renamed CUSTOMER__MIG__202410180318PM.

4.3.2.1.2. Current Table Primary Key Name

PK_{ *base entity name* }

E.g. PK_CUSTOMER would be renamed to PK_CUSTOMER__MIG__202410180318PM.

4.3.2.1.3. Historical Table Name

The table name will have a suffix indicator of MIG added (for migration) and the date and time of the table change and migration.

{ *source table name converted to all caps snake case* }__HIST__MIG__{ *date and time of table evolution* }

{ *date and time of table evolution* } = { *YYYYMMDDHHMM*[*AM* | *PM*] }

E.g. CUSTOMER__HIST would be renamed CUSTOMER__HIST__MIG__202410180318PM.

4.3.2.1.4. Historical Table Primary Key Name

PK_{ *base entity name* }__HIST

E.g. PK_CUSTOMER__HIST would be renamed
PK_CUSTOMER__HIST__MIG__202410180318PM.

4.3.2.2. Data Migration

Data should be migrated for all previously existing columns using an INSERT SELECT. Note there is special processing needed when migrating the __PSTAGE_HASH_DIFF column. Also note the date metadata columns are transferred as is. The dates are not updated in any way to indicate a data change.

4.3.2.2.1. Hash Diff Recalculation

As part of the data migration, the __PSTAGE_HASH_DIFF column will need to be re-calculated for each row in the current and historical table to include the new column. The new column data should be presumed to be null in the calculation and replaced with an empty string, and that empty string needs to be properly positioned in alphabetical order in the calculation.

E.g. Given this was the hash diff calculation prior to the table being changed and a new column named LAST_SOMETHING.

```
MD5(
    IFNULL(TRIM(APPT_DATE), '')
    || '-' || IFNULL(TRIM(APPT_GROUP_ID), '')
    || '-' || IFNULL(CAST(CREATED_BY AS VARCHAR(128)), '')
    || '-' || IFNULL(CAST(CREATE_TIMESTAMP AS VARCHAR(128)), '')
    || '-' || IFNULL(TRIM(DELETE_IND), '')
    || '-' || IFNULL(CAST(MODIFIED_BY AS VARCHAR(128)), '')
    || '-' || IFNULL(CAST(MODIFY_TIMESTAMP AS VARCHAR(128)), '')
    || '-' || IFNULL(TRIM(TO_VARCHAR(ROW_TIMESTAMP, 'BASE64')), '')
) AS _NATIVE_HASH_DIFF,
```

The hash calculation would be this for this initial insert depending on where the new column would be placed.

```
    MD5(
        IFNULL(TRIM(APPT_DATE), '')
        || '-' || IFNULL(TRIM(APPT_GROUP_ID), '')
        || '-' || IFNULL(CAST(CREATED_BY AS VARCHAR(128)), '')
        || '-' || IFNULL(CAST(CREATE_TIMESTAMP AS VARCHAR(128)), '')
        || '-' || IFNULL(TRIM(DELETE_IND), '')
        || '-' || '' --LAST_SOMETHING empty column as positioned
alphabetically in the hash diff.
        || '-' || IFNULL(CAST(MODIFIED_BY AS VARCHAR(128)), '')
        || '-' || IFNULL(CAST(MODIFY_TIMESTAMP AS VARCHAR(128)), '')
        || '-' || IFNULL(TRIM(TO_VARCHAR(ROW_TIMESTAMP, 'BASE64')), '')
    ) AS _NATIVE_HASH_DIFF,
```

4.3.2.3. Table Swap

The new table and the older table should be swapped using the appropriate SQL commands. For example, Snowflake provides the following SQL command to swap tables.

```
ALTER TABLE [ IF EXISTS ] <name> SWAP WITH <target_table_name>
```

E.g. Given the tables CUSTOMER and CUSTOMER__MIG__202410180318PM and CUSTOMER__HIST and CUSTOMER__HIST__MIG__202410180318PM

```
ALTER TABLE CUSTOMER__MIG__202410180318PM
SWAP WITH CUSTOMER;
ALTER TABLE CUSTOMER__HIST__MIG__202410180318PM
SWAP WITH CUSTOMER__HIST ;
```

Other RDBMS would use a more traditional table rename method e.g.:

```
ALTER TABLE CUSTOMER RENAME TO CUSTOMER__TEMP;
ALTER TABLE CUSTOMER__TEMP DROP CONSTRAINT PK_CUSTOMER;
ALTER TABLE CUSTOMER__MIG__202410180318PM RENAME TO CUSTOMER;
ALTER TABLE CUSTOMER__TEMP RENAME TO CUSTOMER__MIG__202410180318PM;
ALTER TABLE CUSTOMER__MIG__202410180318PM RENAME TO CUSTOMER;
ALTER TABLE CUSTOMER ADD CONSTRAINT PK_CUSTOMER PRIMARY KEY (..columns);
```

4.3.2.3. Table Cleanup

The old version of converted table (the one named with the suffix of

__MIG__{ *date and time of table evolution* }

should be dropped by the process after a specified amount of time.

Note: This should not be done for deprecated tables created and noted in the Key Change section. Those need to be preserved.

4.3.2. Current Key Snapshot Tables

There is no migration needed for current key snapshot tables as there are no attributes except primary key attributes so column additions will not occur in this table. If the source key is changed see the source key change section above.

Bibliography

Voss 20190513: Roelant Voss, Why you really want a Persistent Staging Area in your Data Vault architecture, , https://roelantvos.com/blog/why-you-really-want-a-persistent-staging-area-in-your-data-vault-architecture/
Wikipedia 1 n.d.: , Dimensional modeling, , https://en.wikipedia.org/wiki/Dimensional_modeling
Wikipedia 2 n.d.: , Database normalization, , https://en.wikipedia.org/wiki/Database_normalization
Wikipedia 3 n.d.: , Denormalization, , https://en.wikipedia.org/wiki/Denormalization
D'Souza 2015: Jennfer D'Souza, A Multi-Pass Sieve for Name Normalization, 2015
IBM 20250711: Corporate Authors, Schema evolution support, 2025, https://www.ibm.com/docs/en/db2w-as-a-service?topic=tables-schema-evolution-support
MS Learn 1 n.d.: Corporate Author, Transact-SQL syntax conventions (Transact-SQL), , https://learn.microsoft.com/en-us/sql/t-sql/language-elements/transact-sql-syntax-conventions-transact-sql?view=sql-server-ver15&tabs=code