# Model Driven Software Engineering for Data Warehousing — Part 2: Data Warehousing

Everything should be made as simple as possible, but not simpler.

— Attributed to Albert Einstein

This article is the second part of a series of articles in which I want to give an overview of how I think Model Driven Software Engineering (MDSE) can be used for data warehousing.

This article is about todo:DESCRIPTION.

Last but not least the topics described in this article are explained using an example.

todo:INTRODUCTION

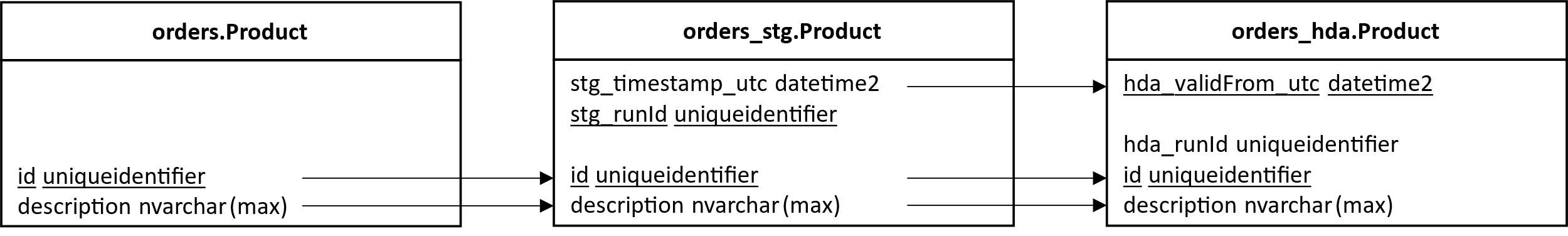
## Generating Data Warehouse Artifacts

Now, we also want to start to work on our actual data warehouse.

At Nippur, the starting point for that is to create historical storage of our data.

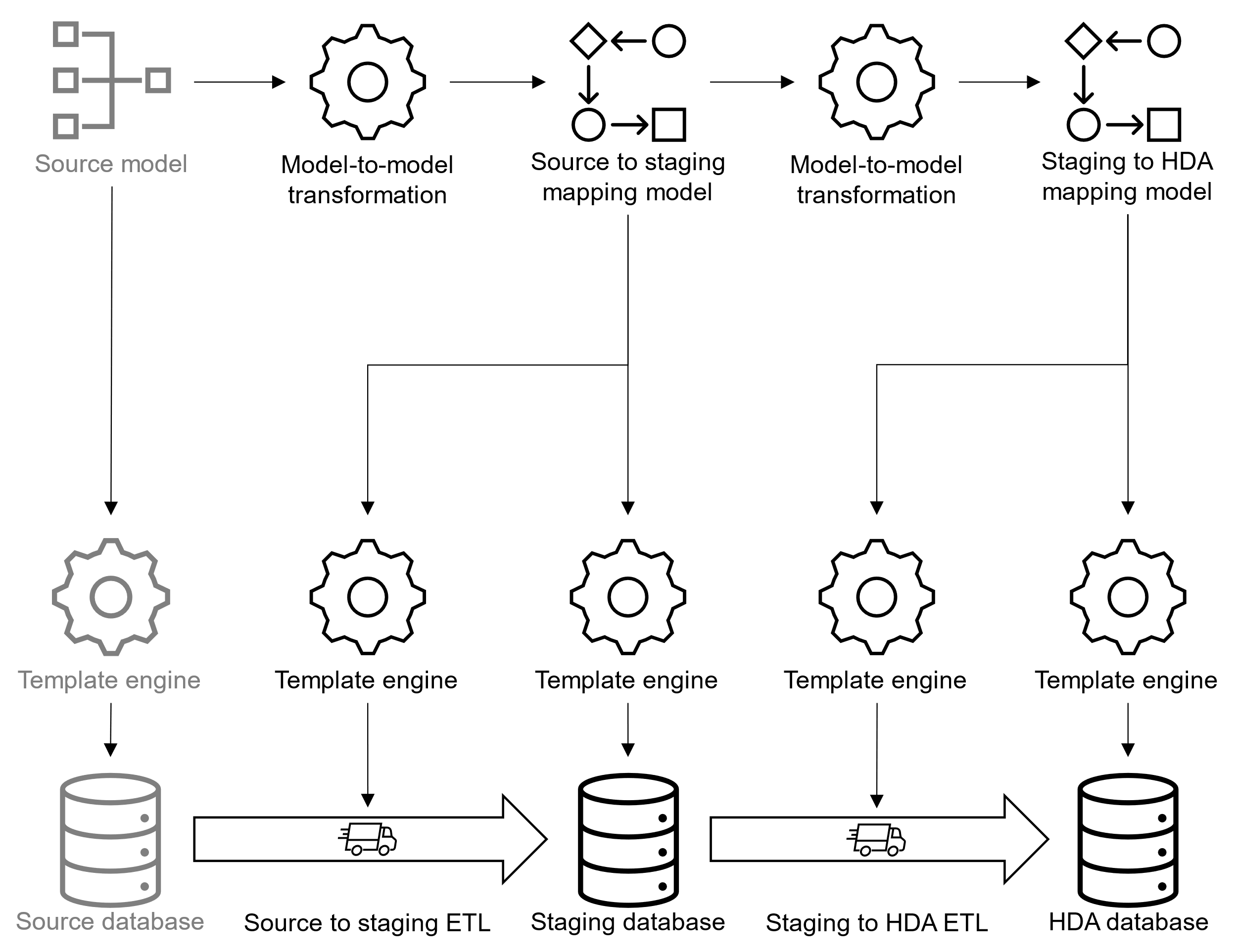
We call this the *Historical Data Archive* (HDA).

To do this, first make sure we can actually store the data in HDA table we transform our previous database model into HDA database model and generated DDL code.



The logic to load staging tables is as follows. Take the @stg\_runId and @stg\_timestamp\_utc as input parameters and inserts data from the [orders].[Product] table into the [orders\_stg].[Product] table, adding the provided staging timestamp and run ID values to each inserted row. This allows for loading and staging data for further processing or analysis in the staging environment.

The logic to load HDA tables is as follows. Take the @hda\_runId and @stg\_runId as input parameters and performs two operations. First, it inserts data from the [orders\_stg].[Product] table into the [orders\_hda].[Product] table, adding the provided HDA run ID, staging timestamp, and other column values. Then, it cleans up the staging table by deleting the rows that were successfully loaded into the HDA table.



## Model transformation for staging tables

The given code defines a set of functions that transform a physical source model into a physical staging model. Here's an explanation of each function:

transform\_pyhical\_source\_model\_to\_physical\_staging\_mapping(source\_model: PhysicalModel) -> PhysicalModelMapping:

This function takes a PhysicalModel object, representing the source model, as input.

It creates a list of staging\_schema\_mappings by applying the function transform\_source\_schema\_to\_staging\_schema\_mapping to each schema in the source model.

It creates a staging\_model by extracting the target schema from each staging\_schema\_mapping.

Finally, it returns a PhysicalModelMapping object that contains the source model, staging model, and schema mappings.

transform\_source\_schema\_to\_staging\_schema\_mapping(source\_schema: Schema) -> SchemaMapping:

This function takes a Schema object, representing a source schema, as input.

It creates a list of staging\_table\_mappings by applying the function transform\_source\_table\_to\_staging\_table\_mapping to each table in the source schema.

It creates a staging\_schema by appending "\_stg" to the name of the source schema and extracting the target table from each staging\_table\_mapping.

Finally, it returns a SchemaMapping object that contains the source schema, staging schema, and table mappings.

transform\_source\_table\_to\_staging\_table\_mapping(source\_table: Table) -> TableMapping:

This function takes a Table object, representing a source table, as input.

It creates a list of staging\_column\_mappings by applying the function transform\_source\_column\_to\_staging\_column\_mapping to each column in the source table.

It creates a list of technical\_staging\_column\_mappings by calling the create\_technical\_staging\_column\_mappings function.

It concatenates technical\_staging\_column\_mappings with staging\_column\_mappings to form column\_mappings.

It creates a staging\_table using the name, primary key constraint, columns, and foreign key constraints derived from the source table and column mappings.

Finally, it returns a TableMapping object that contains the source table, staging table, and column mappings.

create\_technical\_staging\_column\_mappings() -> list[Column]:

This function creates a list of ColumnMapping objects that represent technical staging columns.

It creates two ColumnMapping objects, each containing a Column object representing a target column and an expression.

The target columns have specific names, datatypes, and nullable constraints.

transform\_source\_column\_to\_staging\_column\_mapping(source\_column: Column, source\_table: Table) -> ColumnMapping:

This function takes a Column object, representing a source column, and a Table object, representing the source table, as input.

It creates a staging\_column object with the same properties as the source column, except for the nullable constraint.

The nullable constraint is set to False if the source column is part of the primary key constraint in the source table; otherwise, it is set to True.

Finally, it returns a ColumnMapping object that contains the source column and staging column.

## Generate code to load staging tables

1. {# Define funcionality to quote and join lists of names #}
2. {% **macro** q(names) -%}
3. [{{ names|join('].[') }}]
4. {%- **endmacro** %}
5. {# Define a macro to put commas ',' before items in a loop,
6. except before the first item #}
7. {% **macro** c(loop) -%}
8. {{ ' ' **if** loop.index0 == 0 **else** ', ' }}
9. {%- **endmacro** %}
10. {# Define a macro that returns the value for a column from a column mapping #}
11. {% **macro** colval(mapping) -%}
12. {{ q([mapping.source\_column.name]) **if** mapping.source\_column **else** mapping.expression }}
13. {%- **endmacro** %}
14. {# Loop trough all schema mappings and create SQL ETL statements -#}
15. {% **for** schema\_mapping **in** schema\_mappings %}
16. {# Loop trough all table mappings and create SQL ETL statements -#}
17. {% **for** table\_mapping **in** schema\_mapping.table\_mappings %}
18. -- Load **table** {{ schema\_mapping.target\_schema.name }}.{{ table\_mapping.target\_table.name }}
19. **CREATE** OR **ALTER** PROCEDURE {{ q([schema\_mapping.target\_schema.name, "usp\_load" + table\_mapping.target\_table.name]) }} (
20. @stg\_runId **uniqueIdentifier**
21. , @stg\_timestamp\_utc datetime
22. )
23. AS
24. BEGIN
25. INSERT INTO {{ q([schema\_mapping.target\_schema.name, table\_mapping.target\_table.name]) }} (
26. {% **for** column\_mapping **in** table\_mapping.column\_mappings -%}
27. {{ c(loop) }}{{ q([column\_mapping.target\_column.name]) }}
28. {% **endfor** -%}
29. )
30. SELECT
31. {% **for** column\_mapping **in** table\_mapping.column\_mappings -%}
32. {{ c(loop) }}{{ colval(column\_mapping) }}
33. {% **endfor** -%}
34. FROM
35. {{ q([schema\_mapping.source\_schema.name, table\_mapping.source\_table.name]) }}
36. ;
37. END
38. ;
39. **GO**
40. {% **endfor** %}
41. {% **endfor** %}

## Model transformations for HDA tables

The given code defines a set of functions that transform a staging physical model mapping into an HDA (Historical Data Archive) physical model mapping. Here's an explanation of each function:

transform\_staging\_physical\_model\_mapping\_to\_hda\_physical\_model\_mapping(physical\_model\_mapping: PhysicalModelMapping) -> PhysicalModelMapping:

This function takes a PhysicalModelMapping object, representing the staging physical model mapping, as input.

It creates a list of hda\_schema\_mappings by applying the function transform\_staging\_schema\_mapping\_to\_hda\_schema\_mapping to each schema mapping in the staging physical model mapping.

It creates an hda\_physical\_model by extracting the target schema from each hda\_schema\_mapping.

Finally, it returns a PhysicalModelMapping object that contains the target model from the staging physical model mapping, the hda\_physical\_model, and the hda\_schema\_mappings.

transform\_staging\_schema\_mapping\_to\_hda\_schema\_mapping(schema\_mapping: SchemaMapping) -> SchemaMapping:

This function takes a SchemaMapping object, representing a staging schema mapping, as input.

It creates a list of hda\_table\_mappings by applying the function transform\_staging\_table\_mapping\_to\_hda\_table\_mapping to each table mapping in the staging schema mapping.

It creates an hda\_schema by appending "\_hda" to the name of the source schema from the staging schema mapping and extracting the target table from each hda\_table\_mapping.

Finally, it returns a SchemaMapping object that contains the target schema from the staging schema mapping, the hda\_schema, and the hda\_table\_mappings.

transform\_staging\_table\_mapping\_to\_hda\_table\_mapping(table\_mapping: TableMapping) -> TableMapping:

This function takes a TableMapping object, representing a staging table mapping, as input.

It creates a list of hda\_column\_mappings by applying the function transform\_staging\_column\_mapping\_to\_hda\_column\_mapping to each column mapping in the staging table mapping.

It creates a list of technical\_hda\_column\_mappings by calling the create\_technical\_hda\_column\_mappings function.

It concatenates technical\_hda\_column\_mappings with hda\_column\_mappings to form column\_mappings.

It creates an hda\_table using the name, primary key constraint, columns, and foreign key constraints derived from the target table in the staging table mapping and the column mappings.

Finally, it returns a TableMapping object that contains the target table from the staging table mapping, the hda\_table, and the column\_mappings.

transform\_staging\_column\_mapping\_to\_hda\_column\_mapping(column\_mapping: ColumnMapping) -> ColumnMapping:

This function takes a ColumnMapping object, representing a staging column mapping, as input.

It creates an hda\_column object with the same properties as the target column in the staging column mapping, except for the name.

If the target column name is "stg\_timestamp\_utc", the name of the hda\_column is set to "hda\_validFrom\_utc"; otherwise, it keeps the same name as the target column.

Finally, it returns a ColumnMapping object that contains the target column from the staging column mapping and the hda\_column.

create\_technical\_hda\_column\_mappings() -> list[ColumnMapping]:

This function creates a list of ColumnMapping objects that represent technical HDA columns.

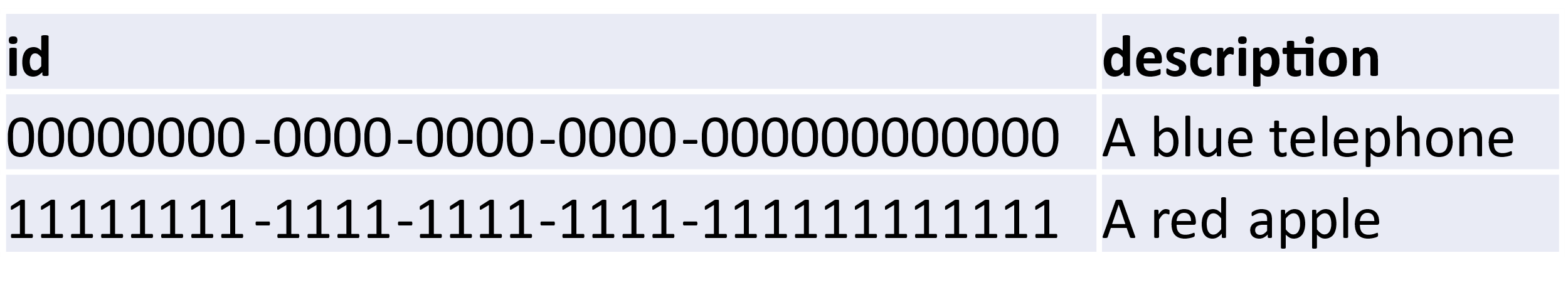
It creates a ColumnMapping object containing a Column object

## Generate code to load HDA tables

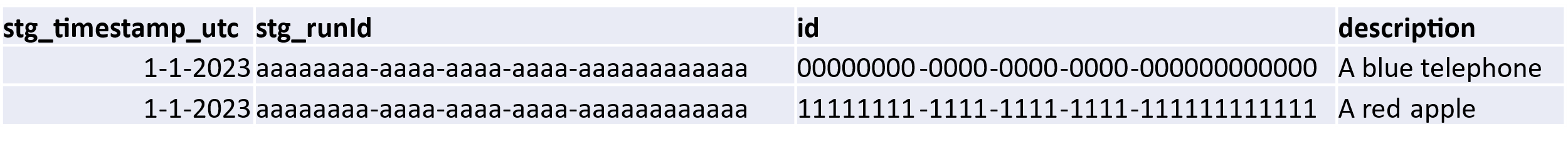
1. {# Define funcionality to quote and join lists of names #}
2. {% **macro** q(names) -%}
3. [{{ names|join('].[') }}]
4. {%- **endmacro** %}
5. {# Define a macro to put commas ',' before items in a loop,
6. except before the first item #}
7. {% **macro** c(loop) -%}
8. {{ ' ' **if** loop.index0 == 0 **else** ', ' }}
9. {%- **endmacro** %}
10. {# Define a macro that returns the value for a column from a column mapping #}
11. {% **macro** colval(mapping) -%}
12. {{ q([mapping.source\_column.name]) **if** mapping.source\_column **else** mapping.expression }}
13. {%- **endmacro** %}
14. {# Loop trough all schema mappings and create SQL ETL statements -#}
15. {% **for** schema\_mapping **in** schema\_mappings %}
16. {# Loop trough all table mappings and create SQL ETL statements -#}
17. {% **for** table\_mapping **in** schema\_mapping.table\_mappings %}
18. -- Load **table** {{ schema\_mapping.target\_schema.name }}.{{ table\_mapping.target\_table.name }}
19. **CREATE** OR **ALTER** PROCEDURE {{ q([schema\_mapping.target\_schema.name, "usp\_load\_" + table\_mapping.target\_table.name]) }} (
20. @hda\_runId **uniqueidentifier**
21. , @stg\_runId **uniqueidentifier**
22. )
23. AS
24. BEGIN
25. -- Load HDA **table** with data from specific staging load.
26. INSERT INTO {{ q([schema\_mapping.target\_schema.name, table\_mapping.target\_table.name]) }} (
27. {% **for** column\_mapping **in** table\_mapping.column\_mappings -%}
28. {{ c(loop) }}{{ q([column\_mapping.target\_column.name]) }}
29. {% **endfor** -%}
30. )
31. SELECT
32. {% **for** column\_mapping **in** table\_mapping.column\_mappings -%}
33. {{ c(loop) }}{{ colval(column\_mapping) }}
34. {% **endfor** -%}
35. FROM
36. {{ q([schema\_mapping.source\_schema.name, table\_mapping.source\_table.name]) }}
37. WHERE
38. {{ q(["stg\_runId"]) }} = @stg\_runId
39. ;
40. -- Cleanup staging **table** after HDA load succeeded.
41. DELETE FROM
42. {{ q([schema\_mapping.source\_schema.name, table\_mapping.source\_table.name]) }}
43. WHERE
44. {{ q(["stg\_runId"]) }} = @stg\_runId
45. ;
46. END
47. ;
48. **GO**
49. {% **endfor** %}
50. {% **endfor** %}

## An example with data

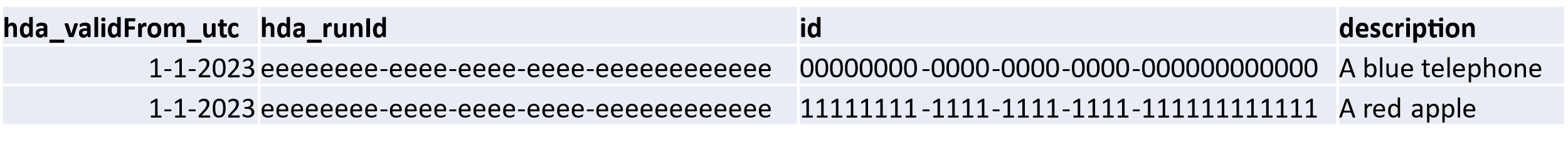
Assume that the first time we want to load the data from our source system, the table [orders].[Product] contains the following records.



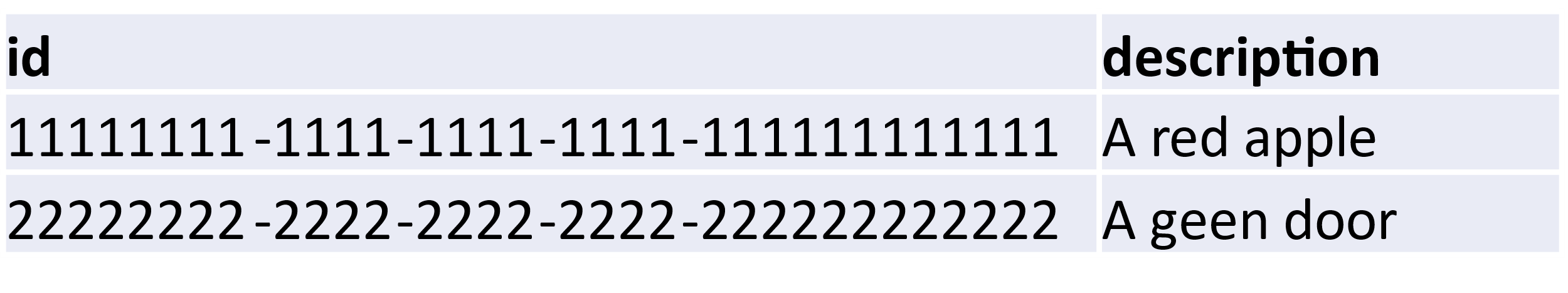
After executing [orders\_stg].[usp\_loadProduct] with parameter values @stg\_timestamp\_utc = '01-01-2023' and @stg\_runId = 'aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa' the table [orders\_stg].[Product] contains the following records.



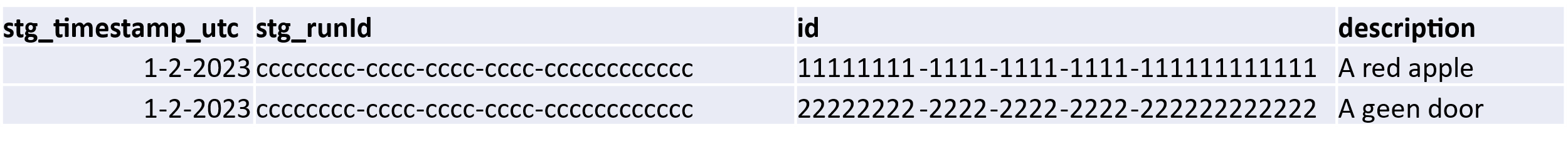
After executing stored procedure [orders\_hda].[usp\_loadProduct] with parameter values @stg\_runId = 'aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa' and @hda\_runId = 'eeeeeeee-eeee-eeee-eeee-eeeeeeeeeeee' the table [orders\_hda].[Product] contains the following records.



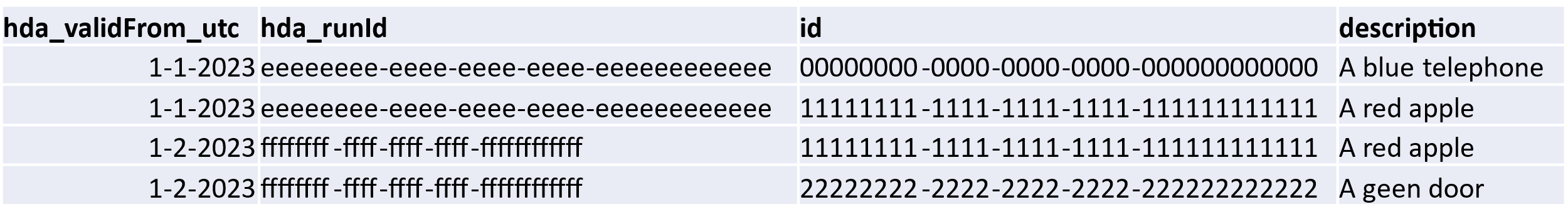
Assume that the next time we want to extract the data from our source system, the table [orders].[Product] contains the following records.



After executing [orders\_stg].[usp\_loadProduct] with parameter values @stg\_timestamp\_utc = '01-02-2023' and @stg\_runId = 'cccccccc-cccc-cccc-cccc-cccccccccccc' the table [orders\_stg].[Product] contains the following records.



After executing stored procedure [orders\_hda].[usp\_loadProduct] with parameter values @stg\_runId = 'cccccccc-cccc-cccc-cccc-cccccccccccc' and @hda\_runId = 'ffffffff-ffff-ffff-ffff-ffffffffffff' the table [orders\_hda].[Product] contains the following records.



## Afterthoughts

One could argue that the historical storage described in this article is not optimal. And I would agree. The goal of this article was to show techniques and considerations when applying MDSE to automate generation of load procedures for data warehouses and I believe that the examples shown in this article are sufficient for that matter.