RAYVENTORY®

Technology Asset Inventory

ETL Implementation Guide 12.5

raynet

ETL Implementation Guide

Raynet GmbH Germany
See our website for locations.

www.raynet.de

Contents

Introduction 5
Basic concepts 6
Technical implementation 7
ETL and Data Hub 9
Tutorial and implementation guide 10
Prerequisites 10
Starting the examples 10
Debug and troubleshoot 12
JSON format 14
Mapping and selecting 17
Single column 19
Multiple columns 21
Defining aggregations 24
Fixed value 25
Auto value 27
Transformed value 29
Regular expression match 31
Switch-case statement 33
Custom SQL statement 35
Advanced topics 37
Fallback columns 37
Union mapping 39
Arbitrary column types 41
Inferring remaining columns 42
Filtering 43
Logical operators 45
Comparison and other operators 46
Custom filtering 47
Grouping and deduplicating 48
Joining 54
Cell merging 55
Overriding cell merging strategy 58
Order of joining 62
Tables priority 62
Wildcard joining 66
Column selection 67

Advanced deduplicating ... 67
Using MaxValue and MinValue strategy ... 69
Splitting ... 69
Enriching ... 71
Look-up values ... 74
Split look-up ... 74
Look-up keys ... 76
Fallback ... 77
Consolidating look-up targets ... 77
Chaining steps ... 78
Optional and required tables ... 80
Programmability ... 81
SQL environment ... 81
Creating reusable scripts ... 81
Extra functions ... 82
Additional Information ... 84

Introduction

With the new ETL technology, Raynet is on the way to bring the groundbreaking vision of a fully automated, UI-supported and use case-tailored data transformation to life. With the current version, it is possible to implement the transformation process in a configurable way, even without programming knowledge. At the same time, the ETL process is documented in a structured and comprehensible way.

Basic concepts

In order to transform the data (Transform), we need to plug-in some initial raw or partially transformed data (Extract), and the after processing load it back to the data source (Load).

Transformation step should be understood as a process of one or more steps, which may be interconnected, with many inputs and many outputs. The following steps are currently available:

- **Map**
  Transforms one table to another table. Mapping can take existing columns as-is, rename them, perform simple or complex transforming, aggregating or disaggregating them etc. The result table has always as many rows as the input table, but the number of columns may be different.

- **Filter**
  Takes the input table and filter out items that do not match a specific set of conditions. The result table has always the same number of columns, but may have a different number of rows, depending on the filtering.

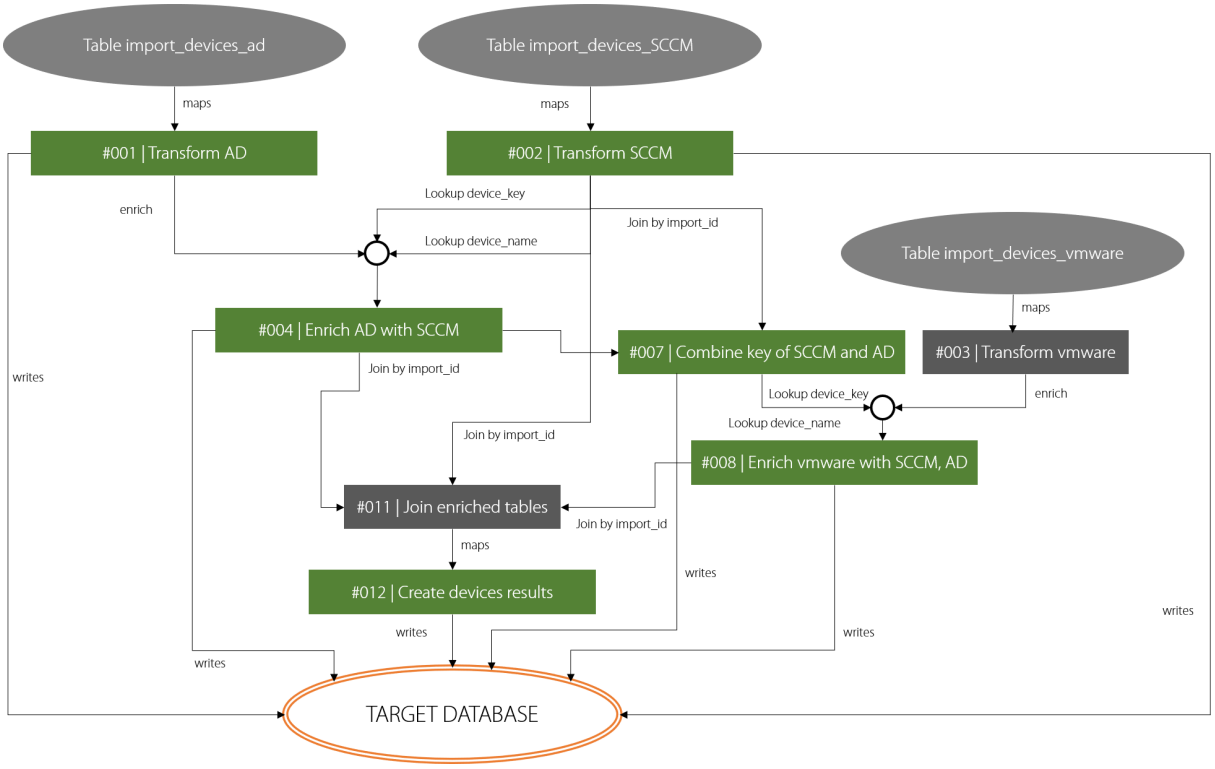
- **Join**
  Joins two or more tables together by combining columns with the same names, using specific methods of aggregation. The input are two or more tables, and the output is a single table.

- **Deduplicate**
  Based on unique value of one or more columns, this steps deduplicates the table based on the conditions defined by the user.

- **Enrich**
  Adds information to a specific column of a source table by performing a look-up in another table(s) using set of ordered look-up rules. The input is a single table + a set of look-up tables, the output is a single table with all the columns from the source table + the enriched column if it was not present in the original table.

These steps can be used to perform complex operations by splitting big unit of works into smaller bits, that are building blocks for the complex transformation.

This sample advanced process may have the following workflow:



Transformation process is a chunk of steps which perform a given task (for example: take Active Directory, RayVentory and Vmware source raw data and produce a single table combining the sources together). Obviously, there may be more independent ETL processes defined for a single data set.

As pictured above, a single step (for example a single mapping or joining) can have different number of sources, and itself it may be a source for one or more steps.

The results of each step is by default temporary that means it will not be written to the target database. To mark a step as "result", the user must do it explicitly. In the chart above, all "green" steps are written to the target database, but the gray ones (#011 and #003) are not and exist only as intermediary steps for other steps.

An output of a step if always a single table, which can be temporary or permanent (written back to the database). However, it is important not to confuse the step with the table. Each step is an operation on the data, which has inputs and output, and the output of the step is a table.

Technical implementation

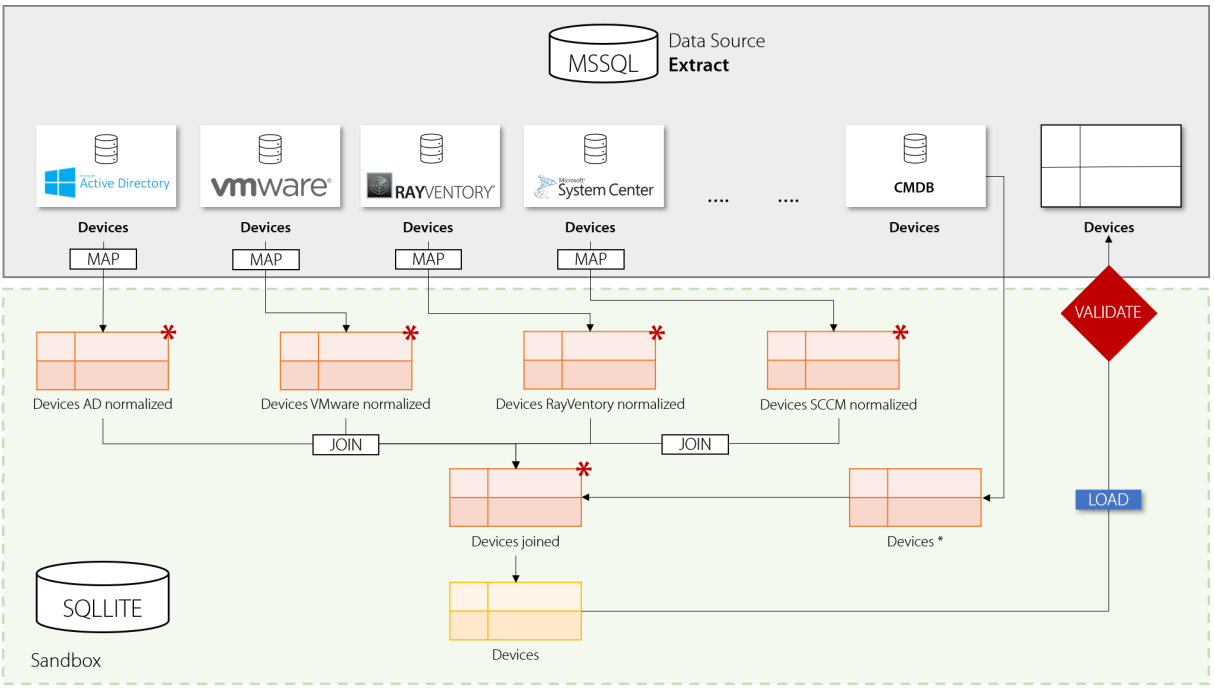
ETL library by Raynet has the following principles:

- Cross-platform (works on Windows, UNIX, Docker etc.)
- Independent from data source (currently supports MS SQL, but is platform-agnostic in its core. Future implementations will include SQlite, MariaDB, OracleDB, REST etc.)
- Secure (all operations happen in a sandbox)
- Transparent (instead of large, complex operations, the work is split to smaller units of work)
- Efficient (works with data sets having several terabytes of sizes)
- Portable (has no dependencies to any DB and other frameworks, except of the data that is extracted from source and loaded to the target)
- Configurable (all operations are defined in JSON format, no direct SQL involved)
- Flexible and easy to change (complex operations are abstracted, so that typical tasks can be achieved with a single change, without writing complex join or other queries).

To achieve this, the following is involved:

- The user defines his transformation processes in a JSON file with a specified syntax. This JSON file can be crafted by hand or by using some visual editing tools that Raynet provides in the future (see section tooling).
- The transformation layer performs extraction by reading out the necessary tables that the user required. It then copies these tables from the data source (with help of specific data adapter) to a sandbox environment, utilizing embedded temporary SQLite database.
- The transformation is performed in the SQLLite database and has no access to the original data source. All steps can see only the tables that were referenced by the user.
- Once the transformation is done, the tables from the sandbox are copied back to the target environment, again using specific data adapter.

The operations executed in each step may be executed either directly in the sandbox (native SQLLite query) or in memory. The engine tries to rely as much as possible to the sandbox database engine, which is usually the fastest way of transforming, filtering etc. Using certain features provided by the engine may transparently switch to in-memory execution, which may involve small performance hit.



At the end, data validation should happen on the RayVentory Data Hub site, to ensure that the tables are correct from the business point of view (all columns, required names, non-null fields etc.). This does not belong to the scope of ETL library from Raynet.

The operation that happen in the sandbox may be parallelized. By default, the library tries to create a graph of dependencies and execute tasks in batches, if all dependencies are fulfilled. In the example above, first four tasks *DevicesAD_Normalized*, *DevicesVmware_Normalized*, *DevicesRayVentory_Normalized* and *DevicesSCCM_Normalized* have only dependencies to the source DB (and no other steps), so they may be executed in parallel. On the other hand, there are some other steps that depend on these four, so the other steps will wait until all four are executed.

ETL and Data Hub

Data Hub adds an extra layer on top of ETL library when used together. The hub takes over the responsibility of data extraction, and does it from much more sources than ETL library can (for example cloud connectors, Active Directory, PowerShell etc.).

The Extraction part in the ETL library works then directly on the already extracted data. It ensures, that the transformation and all required operations are executed in a clean sandbox.

- **Extract** -> should be understood as reading the values from the RayVentory Data Hub Results table – data already collected by any Data Agent
- **Load** -> should be understood as writing back normalized result(s) to the RayVentory Data Hub Results table.

Extraction and loading can be implemented in the hosted instance (task type transformation) or on agent site.

Tutorial and implementation guide

Prerequisites

- To edit the JSON, it is assumed that the reader has a knowledge of JSON notation. Use some decent JSON editor to help you editing the files to avoid typos and other mistakes
- It is not required but still highly recommended to have basic knowledge of SQL (any flavour) to understand some important concepts

Starting the examples

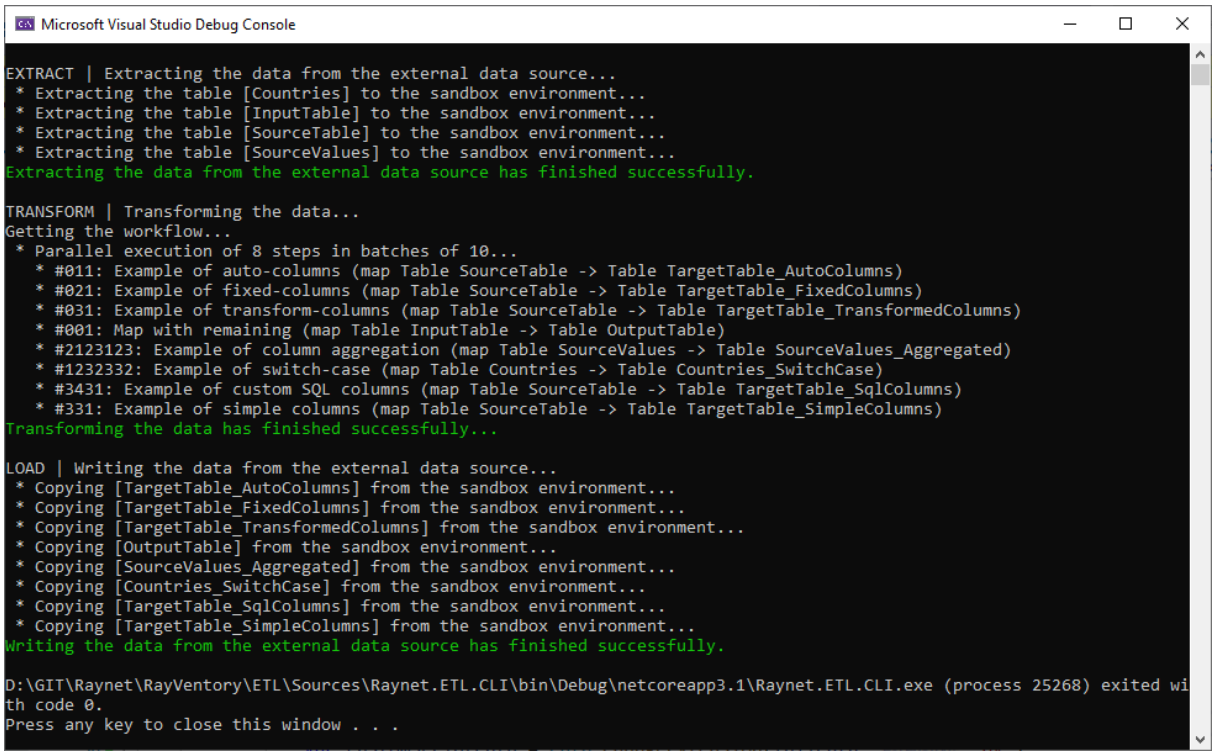
For a quick start with the console, an MS SQL database with some source tables is required. The samples below show how to hand-write the JSON definition. After saving the file in JSON format, you can run it using the console:

```
Raynet.Etl.Console.exe -f "<path-to-json>" -c "<connection-string-to-mssql-database>"
```

For example:

```
Raynet.Etl.Console.exe -f "c:\temp\def.json" -c "Server=.
\SQLEXPRESS;Database=RayVentoryDataHubResult;Trusted_Connection=True;"
```

The console application outputs information about the performed job, for example:



Partial samples and multiple files

Note:
These options are available in ETL module version 1.1.271 and newer.

It is possible to split the definition across many JSON files, which are then joined by the ETL runtime. Partial definitions may reference steps from other files, and they will be correctly resolved and processed in parallel by the engine.

To use partial JSON definition you can provide multiple arguments to the `-f` switch, for example:

```
Raynet.Etl.Console.exe -f "c:\temp\def-1.json" "c:\temp\def-2.json" -c "Server=.
\SQLEXPRESS;Database=RayVentoryDataHubResult;Trusted_Connection=True;"
```

Alternatively, if all JSON files lie in the same folder and have .json file extension, then instead of the path to the file, the full path to the directory may be provided instead:

```
Raynet.Etl.Console.exe -f "c:\temp\" -c "Server=.
\SQLEXPRESS;Database=RayVentoryDataHubResult;Trusted_Connection=True;"
```

These options can be joined, for example a folder and specific files.

Beware the following limitations:

- You must ensure the files can be merged by having unique step IDs across all files.

- ETL engine does not de-duplicate your resources - if a file is referenced twice (whether by file path or a folder path) it will be merged twice.
- If using more than one input file, there is no automatic SQL and PY file detection. In case of a single file, they are resolved automatically by taking the file name with a proper extension. In multi-file scenario, you should specify the value of command line parameter to control which script and macro definition gets loaded.

Debug and troubleshoot

There are some different techniques for debugging and troubleshooting ETL.

Verbose output

By default, only some basic information and success/failure messages are logged in the console window. To activate a more verbose output, add `--verbose true` switch to your command line. Verbose mode is activated automatically if any of `--debug` or `--break` command line switch is used.

In verbose mode, the following features are activated

- **More verbose messages**
  More info is logged in the console. Additional information include: duration of operations, files being used (database file, SQL scripts)
- **Data**
  First three rows of each table output are printed into the console, so that the user can control the execution state.
- **Colum types**
  Column types for tables written back to the DB (load phase) are written into the console output.

Debug mode

Debug mode disabled some optimizations. To use it, add `--debug true` to your command line. Debug mode automatically activated verbose output. The following features are activated in the debug mode:

- **Disabled flow optimizations**
  Without the debug mode, ETL tries to organize the steps in batches and executes up to 10 steps in parallel, ensuring that the dependencies are fulfilled. In the debug mode, the steps are ordered to fulfill the dependencies and requirements, but are always started sequentially. This means that the execution takes more time, but gives the user a way to control the execution and state after each step.
- **Limited parallelization**
  Some operations are executed sequentially and not in parallel.

- All features introduced by verbose mode.

Breakpoints

Breakpoints are meant to pause the execution after each step. The execution is only continued once the user presses any key. Breakpoints automatically enable verbose output and debug mode. To activate the breakpoints, add `--break true` to your command line. The following features are activated in the breakpoint mode:

- **Breakpoints**
  After each step, a breakpoint is triggered. The execution continues to the next step only after pressing any key. Again, this gives the user some time to check the current step of the database and the transformed data.

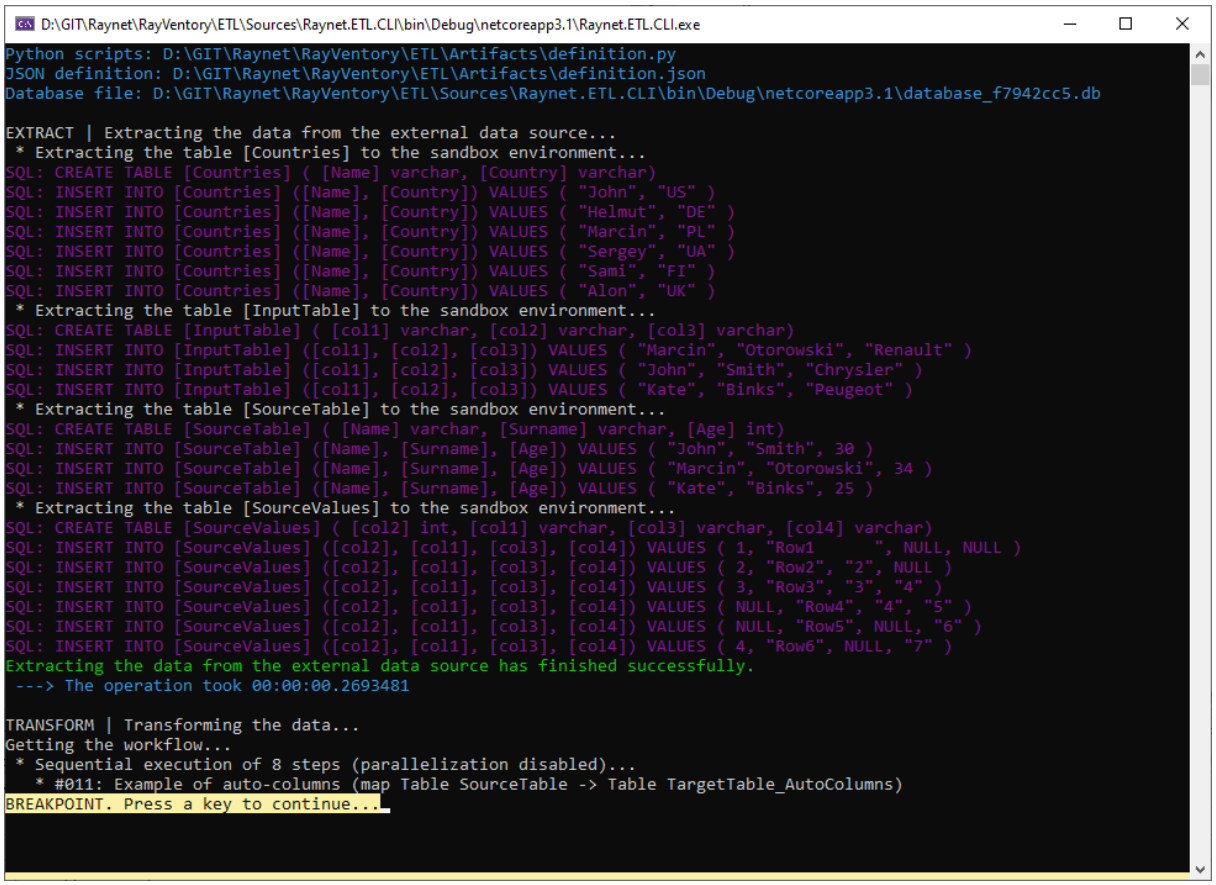
- All features introduced by debug and verbose mode.

Dry mode

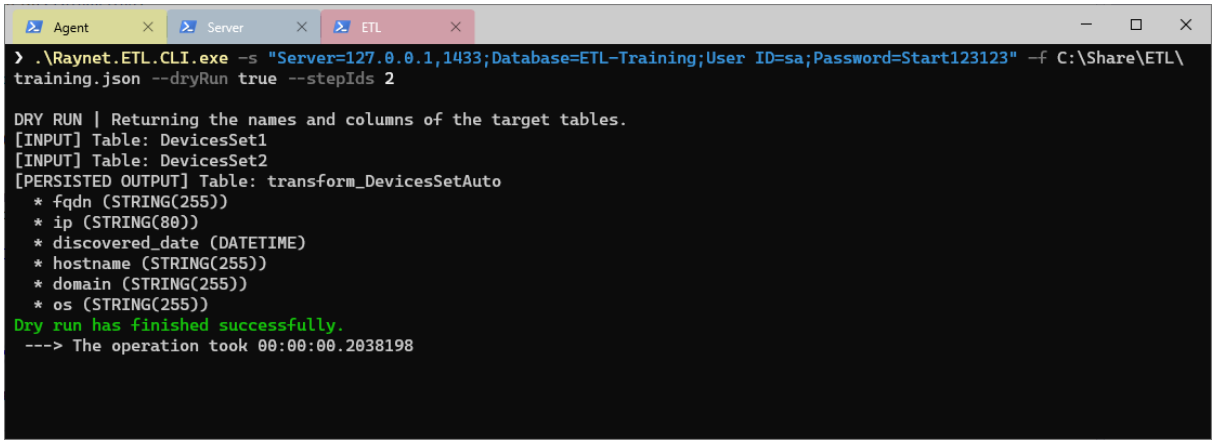
Dry mode enables you to see which tables are required and produced by a given JSON file, without actually doing the transformation. This process is fast as only minimal set of metadata gets pulled from the database. To activate dry mode, add `--dryMode true` to the command line. If the switch is present, you can also use another parameter `--stepIds` which may contain space-separated list of step identifiers to be analyzed. If no step ID is provided, then all steps from the selected JSON file will be analyzed.

Sample output

Sample output from the console when all three switches are used:



Example of dry mode:



JSON format

The tasks and their relations are described using a JSON format. The ETL CLI expects the user to provide a full path to a JSON file for further processing.

This user guide assumes that the user has already an experience with JSON format. Below is a quick recap of the syntax:

Basic data types

- **Number**: An integer or floating-point number. The numbers are written directly without any enclosing quotes, for example `100` or `10.23`.
- **String**: Represents an Unicode text. Strings must be enclosed with opening and closing quotation-marks or a pair of apostrophes. To use a quotation inside a string, it must be preceded by so-called "escape character" - a backslash (\). Valid example of strings are: `""` (empty string), `"Hello World"`, `"Hello \"World\" I have nested quotes"`. Note: Instead of quotation, a single apostrophe can be used - in this case quotation marks do not need to be escaped (on the other hand the apostrophes should). Examples using single apostrophe are: '' (empty string), `'Hello World'`,`'Hello "World" I have nested quotes'`,`'Hello I am John\'s computer'`.
- **Boolean**: Either `true` or `false` (without quotation or apostrophes).
- **Array:** An ordered list of objects of other types (numbers, strings, boolean values, objects or other arrays). Array elements are separated by a comma, and surrounded with a pair of opening and closing square brackets. Valid examples of an array are: [] (empty array), `[ "single string" ]`,`[ 1, 2, 3 ]`,`[ true, 1, "string]` etc.
- **Object**: A collection of name–value pairs where the names are strings. Objects are delimited with curly brackets and use commas to separate each pair, while within each pair the colon ':' character separates the key or name from its value. Examples of valid objects are: `{ }` (empty object), `{ "name": "Marcin" }`, `{ "Name": John", "Surname": "Smith", "Age": 30 }` etc.
- **null**: an empty value, using the word `null`.

Object values and array elements can be of any type, including other objects, arrays or null values.

ETL definition format

In its simple form, the definition of a transformation is a JSON object, which defines an array of steps ("steps"):

```
{
    "steps": [
        {
            // step 1 here
        },
        {
            // step 2 here
        }
        // and so on...
    ]
}
```

JSON does not define the source connection or a database type. Instead, this information is provided via the command-line (when executed as a standalone tool), or passed by DataHub (when executed as a transformation task).

The array of steps can contain one or more steps, available from the list of supported operations

- **Mapping and selecting**
  Transforms one table into another (the number of rows stays the same) or concatenates two or more tables into a single table.

  Similarities to SQL:
  For a single table usage this translates roughly to `SELECT <columns> FROM <source>`
  For multiple tables usage, this translates roughly to `SELECT <all-columns> FROM <source1> UNION ALL SELECT <all-columns> FROM <source2>...`

- **Filtering**
  Takes an input table and produced another one with the same columns, but having less or equal number of rows (based on filtering condition).

  Similarities to SQL:
  This method translates roughly to `SELECT * FROM <source> WHERE <condition>`

- **Grouping and deduplicating**
  Take a table and group the rows based on a matching combination of a single or more columns. Grouping has two modes: it can either only mark the rows which should be grouped by introducing an extra column, or perform de-duplication. De-duplication always takes a single row from each group. Some typical aggregation modes are provided (maximum, minimum, average, concatenation and coalescing).

  Similarities to SQL:
  Grouping itself is similar to `SELECT * FROM <source> GROUP BY <group_key>`. Introducing an extra column with group ID has no direct equivalent.

- **Joining**
  Given a list of several tables, this operation joins them together and automatically resolves conflicting columns using aggregation methods. It provides a way for defining a "master joining order" and override this on column-basis, ensuring that different table may have different priorities in different columns.

  Similarities to SQL:
  Less complex scenarios can be represented by a sequence of JOIN (LEFT, INNER, OUTER) queries, more complex have no simple mapping. Additionally, it is possible to join on wildcards (for example all tables matching the pattern `MyTable*`) and also define priorities based on wildcard patterns. Neither of these is possible with simple SQL.

- **Advanced deduplicating**
  Given some input table, deduplicates it based on several conditions and produces a table without duplicates. The duplicates can be either picked from existing rows or created based on all grouped values. It is also possible to select any number of rows (for example only first, none, all or all but not last etc.)

  Similarities to SQL:

There is nothing which maps 1-1 to this, the de-duplication is represented by several operations involving the usage of `GROUP BY` and aggregations.

Note: This function is powerful but for most cases may be a bit of overkill. A similar functionality is offered by the **Groupping**, which should be the first choice for de-duplication or grouping.

- **Splitting**
  Given a single input table having a column with aggregated values separated by a specific separator, this produces a table which "unpivots" the data.

  Similarities to SQL:
  There is nothing which maps 1-1, the operation is represented by several functions and select queries.

- **Enriching**
  Given a single input table, add extra information to it from other tables, using different methods of joining by different keys.

  Similarities to SQL:
  There is nothing which maps 1-1 to this, enrichment is represented by many different join operations, grouping, sub-queries etc.

The steps can be chained.

Comments

Both single line comments (starting with `//`) and block comments (starting with `/*` and ending with `*/`) are supported in the JSON file.

Mapping and selecting

Mapping is a process which covers one of the following use cases:

- Having a single input table, an output table is created. The new table has the exact number of rows as the first table, but the columns may be different.
- Having several input tables, a single output table is created. The new table has the exact number of rows as all selected tables together, and contains all unique columns from the selected tables. This means that the mapping process can concatenate tables, ensuring that all columns are taken over.

Mapping step requires that the user defines the list of columns to be written in the new table. There are three ways to do it:

- By specifying all required columns

- By skipping the specification of required columns but setting the attribute *mapRemaing* to *true*.
- By using both together - specifying only columns which will be transformed, and using *mapRemaining* to infer the remaining, undefined columns and include them as well.

For more information about map inferring remaining columns to map, read the following chapter: **Inferring remaining columns**.

A column may be taken-over or transformed. The following mappings are available:

- **Simple mapping**
  A column may be simply taken over without any additional processing (simple mapping). The name may be taken as-is or changed to an arbitrary name.
- **Fixed values**
  A new column may be created, containing fixed values.
- **Auto values**
  A new column may be created, containing values inserted dynamically (random numbers, date and time, GUIDs, placeholders).
- **Transformed values**
  A new column may be created by transforming the existing column using a set of transform options (uppercasing, lowercasing, switch-case statements etc.)
- **Aggregated values**
  A new column may be created by aggregating two or more other columns (max/min value, average, concatenated string, first not-null value etc.)
- **Custom values**
  A custom value may be calculated using SQL syntax.

Mapping uses the following syntax:

```
{
  "id": 1, // unique ID
  "type": "map",
  "name": "Description of the step",
  "source": "Name of the source table",
  "columns": {
    // a dictionary of columns
    "TargetName1": {}, // definition of source1
    "TargetName2": {}, // definition of source2
    [...]
  },
  "target": "The name of the output table"
}
```

At minimum the following properties are required:

- ID (must be unique)
- Type (must be set to "map")
- Source (must be one of the following):
  - A string representing the table name.
  - A string representing a wildcard to look for table(s).
  - An integer representing the source as another step.
  - An object with property `table` set to the name of the source table.
  - An object with property `step` set to the ID of the source step.
  - An array of tables or steps to perform union select.
- Either a non-empty list of columns, or the attribute *mapRemaining* set to *true*.

Note:

Note: The target is optional. If omitted, the output table is saved temporarily and - unless not used by any other step - will be discarded once the transformation is finished.

Source is by default optional, meaning that the table does not have to be existing. If the table is missing, the step will not be executed and its target table will not be written. There is a way to define that the source is required, in which case in case of a missing source table the step will fail and report an error. More information about setting up required steps can be found in the following chapter: Optional and required tables.

Value types

The value type for each column is inferred from the context. To enforce an arbitrary column type (for example `VARCHAR(128)`) use the property *type*. More information about arbitrary types can be found in chapter Arbitrary column types

Single column

Simple mapping takes a source column and produces another column with a given target name.

The full syntax for the simple mapping is

```
"target-column-name": {
  "column": "required-source-column-name"
}
```

or

```
"target-column-name": {
  "column": "source-column-name",
  "type": "sql-type"
```

```
}
```

If type is omitted, the type of the column is inferred from the usage (either from parent table, or from the actual value type like string, number etc.).

You should specify the type if:

- The column in the target table should have a specific type or length - for example `varchar(100)` and not `varchar(max)`
- The column in the source table should be cast to another type - for example to treat integer as string etc.

However, if the source type is sufficient, it is possible to use a shorter syntax for the column mapping:

```
"target-column-name": "source-column-name"
```

The names of target columns must be unique. The names of source columns do not have to be unique.

The columns that you do not define will not be written in the target table, unless you instruct the ETL engine to do otherwise by setting the value of *mapRemaining* to *true*.

Example

Given the following table **SourceTable:**

Name	Surname	Age
Marcin	Otorowski	34
John	Smith	51
Kate	Binks	25

And the following JSON step definition:

```
{
  "id": 1,
  "type": "map",
  "name": "Example of simple columns",
  "source": "SourceTable",
  "columns": {
    "FirstName": "Name",
    "LastName": "Surname",
    "Age": "Age",
  },
  "target": "TargetTable_SimpleColumns"
}
```

The following result table **TargetTable_SimpleColumns** is expected:

FirstName	LastName	Age
John	Smith	30
Marcin	Otorowski	34
Kate	Binks	25

Arbitrary types, casting and converting

Using the `type` property lets you define a custom cast or custom type length. This common property is available for all column types, see more information in chapter Arbitrary column types.

Fallback values

It is possible to define a custom value returned, if the source column does not exist in the table. For more information about the usage of fallback values, see the Fallback columns chapter.

Multiple columns

It is possible to calculate the value of a column based on the two or more other columns.

The full syntax for the multiple column mapping is

```
    "target-column-name": {
      "columns": ["required-source-column-name-1", "required-source-column-name-
2"...],
      "aggregation": string|object
    }
```

or

```
    "target-column-name": {
      "columns": ["required-source-column-name-1", "required-source-column-name-
2"...],
      "aggregation": string|object
      "type": "sql-type"
    }
```

When the property `aggregation` is omitted, the default concatenation with pipe (`|`) as a a separator will be used.

Aggregation may be defined either as a string or as an object. For more information about the syntax and supported aggregation methods, see chapter Defining aggregations.

> **Note:**
> Multiple column mapping support basic aggregations, like coalescing, selecting average/minimum/maximum values or concatenating the strings. You can use more complex values and cover more complicated cases by using custom mapping via SQL.

In this example, value from *column2*, *column3* and *column4* will be concatenated using | as the separator. This is the default behavior should no other options be specified. You can instruct ETL layer to use specific algorithm by using object notation:

```
{
    "steps": [
        {
            "id": 2,
            "name": "Normalize sample",
            "type": "map",
            "source": "sample2",
            "target": "sample2_Normalized",
            "columns": {
                "column1": "column1",
                "column2_together":{
                    "columns": ["column2", "column3", "column4"],
                    "aggregation": "coalesce"
                }
            }
        }
    ]
}
```

This definition will select first non-null value from the list, starting from the left to right. To concatenate strings, replace *coalesce* with *concat*. If you need a custom separator, use again an object notation for aggregation and specify it there.

> **Note:**
> Bear in mind that *null* values are omitted from the result of *CONCAT*. This is a handy feature for table join, where you want to track the source table. Join aggregation rules are the same as column rules for mapping.

Example

Given the following table **SourceValues:**

col1	col2	col3	col4
Row1	1	<null>	<null>
Row2	2	2	<null>

col1	col2	col3	col4
Row3	3	3	4
Row4	*<null>*	4	5
Row5	*<null>*	*<null>*	6
Row6	4	*<null>*	7

And the following JSON step definition:

```
{
  "id": 2,
  "name": "Example of column aggregation",
  "type": "map",
  "source": "SourceValues",
  "target": "SourceValues_Aggregated",
  "columns": {
    "col1": "col1",
    "coalesce": {
      "columns": [ "col2", "col3", "col4" ],
      "aggregation": "coalesce"
    },
    "max": {
      "columns": [ "col2", "col3", "col4" ],
      "aggregation": "max"
    },
    "concat_default": {
      "columns": [ "col2", "col3", "col4" ]
    },
    "concat_custom": {
      "columns": [ "col2", "col4", "col3" ],
      "aggregation": {
        "type": "concat",
        "separator": ","
      }
    }
  }
}
```

The following result table **SourceValues_Aggregated** is expected:

col1	coalesce	max	concat_default	concat_custom
Row1	1	1	1	1
Row2	2	2	2 | 2	2,2
Row3	3	4	3 | 3 | 4	3,4,3

col1	coalesce	max	concat_default	concat_custom
Row4	4	5	4 | 5	5,4
Row5	6	6	6	6
Row6	4	7	4 | 7	4,7

Arbitrary types, casting and converting

Using the `type` property lets you define a custom cast or custom type length. This common property is available for all column types, see more information in chapter Arbitrary column types.

Defining aggregations

Aggregations are methods, that - given a sequence of multiple input columns - produce a single value.

In current implementations, aggregations are used:

- by **multiple column mappings** to define how the values from two or more columns are to be transformed into a single column
- by **join mapping**, where aggregation defines the way of solving conflicts between similar columns in two or more tables.

The library supports the following aggregation methods:

- **Avg**
  Selects the average value of one or more values
- **Max**
  Select the maximum of one or more values
- **Min**
  Select the minimum of one or more values
- **Sum**
  Select the sum of one or more values
- **Coalesce** (alias `FirstNonNull`)
  Select first not-empty value from the list of one or more values (from left to right)
- **Concat**
  Join all given non-empty values using a specific separator (from left to right). For this operation, a separator may be defined (default is a pipe character `|`).
  Bear in mind that this operation omits NULL values.

In places where the aggregation is expected, a simple string (one of the values from the table above) is sufficient to define the aggregation, the values are case insensitive. For example, this is how column mapping defines that the maximum of three values should be taken:

```
"target-column-name": {
  "columns": ["column1", "column2", "column3"],
  "aggregation": "max"
}
```

The string syntax is a shorthand form of the full object notation, which would have the following syntax:

```
"target-column-name": {
  "columns": ["column1", "column2", "column3"],
  "aggregation": {
    "type": "max"
  }
}
```

Both forms are functionally equal. You can use either of these for all operations. However, to use the **Concat** method with a custom separator, the object notation must be used. The separator is defined by the `separator` property:

```
"target-column-name": {
  "columns": ["column1", "column2", "column3"],
  "aggregation": {
    "type": "concat",
    "separator": ";"
  }
}
```

Fixed value

You can use a fixed value for a column (for example a string, number or boolean).

To do this, use the following syntax:

```
{
    "columns": {
        "target_column_name":
        {
            "value": "some value"
        },
        [...]
    }
}
```


Note:

Avoid mistakes of mixing this syntax with a similar looking `"target_column_name": "source_column_name"`. The former will use a fixed value `"source_column_name"`, the latter will try to look-up the column named `"source_column_name"`.

Example

Given the following table **SourceTable:**

Name	Surname	Age
Marcin	Otorowski	34
John	Smith	51
Kate	Binks	25

And the following JSON step definition:

```
{
  "id": 1,
  "type": "map",
  "name": "Example of fixed-columns",
  "source": "SourceTable",
  "columns": {
    "Name": "Name",
    "Fixed_String": { "value": "MyValue" },
    "Fixed_Number": { "value": 23 },
    "Fixed_Boolean": { "value": true }
  },
  "target": "TargetTable_FixedColumns"
}
```

The following result table **TargetTable_FixedColumns** is expected:

Name	Fixed_String	Fixed_Number	Fixed_Boolean
Marcin	MyValue	23	1 *(bit)*
John	MyValue	23	1 *(bit)*
Kate	MyValue	23	1 *(bit)*

Arbitrary types, casting and converting

Using the `type` property lets you define a custom cast or custom type length. This common property is available for all column types, see more information in chapter Arbitrary column types.

Auto value

You can use an automatically generated, non-static value for a column. Currently there are 4 types of auto-columns supported:

- **Guid**
  Random UUID (globally unique)
- **Number**
  Autoincremented number (locally unique)
- **DateTime**
  Date and time (not guaranteed to be unique, neither globally nor locally)
- **Date**
  Current date (without time part)
- **Time**
  Current time
- **TableName**
  The name of the source table (not unique). This type of auto-column makes sense in case when two or more tables are selected by a single map statement, and the information about the source table name is important.

To define auto-column, an object notation in the following format is required:

```
{
    "columns": {
        "target_column_name":
        {
            "auto": "<type>"
        },
        [...]
    }
}
```

Where `<type>` is one of the supported value: `Guid`, `Number`, `DateTime` or `TableName`. There are no more properties required or defined by auto-columns.

Example

Given the following table **SourceTable:**

Name	Surname	Age
Marcin	Otorowski	34
John	Smith	51

Name	Surname	Age
Kate	Binks	25

And the following JSON step definition:

```
{
  "id": 1,
  "type": "map",
  "name": "Example of auto-columns",
  "source": "SourceTable",
  "columns": {
    "Name": "Name",
    "Auto_DateTime": { "auto": "datetime" },
    "Auto_Guid": { "auto": "guid" },
    "Auto_Number": { "auto": "number" },
    "Auto_TableName": { "auto": "tableName" }
  },
  "target": "TargetTable_AutoColumns"
}
```

The following result table **TargetTable_AutoColumns** is expected:

Name	Auto_DateTime	Auto_Guid	Auto_Number	Auto_TableName
Marcin	2021-03-17 11:54:40.000	57992940-6633-4528-9B3A-47563855EE14	1	SourceTable
John	2021-03-17 11:54:40.000	FDE0EA32-E252-408F-8743-52A53BD2F141	2	SourceTable
Kate	2021-03-17 11:54:40.000	0E266619-6689-4A2C-B94F-934E6B095E50	3	SourceTable

Arbitrary types, casting and converting

Using the `type` property lets you define a custom cast or custom type length. This common property is available for all column types, see more information in chapter Arbitrary column types.

Transformed value

The value can be transformed using a built-in transform. The following transformation are available:

Transform name	Behavior
Lowercase	Returns lowercase variant of the value. This transformation by default preserves the original data type.
Uppercase	Returns uppercase variant of the value. This transformation by default preserves the original data type.
IsNullOrEmpty	Returns 0 is the value is not null or empty or 1 otherwise. This transformation maps by default to a boolean value.
IsNotNullOrEmpty	Returns 1 is the value is not null or empty or 0 otherwise. This transformation maps by default to a boolean value.
Length	Returns the length of the string value. This transformation maps by default to a 32-bit integer value.
MD5	Hashes the given column with MD5 hash. Returns 128-bit (16 bytes) hexadecimal string of 32 characters. *Note: This transformation is available in ETL module version 1.1.271 and newer.*
SHA1	Hashes the given column with SHA1 hash. Returns 160-bit (20 bytes) hexadecimal string of 40 characters. *Note: This transformation is available in ETL module version 1.1.271 and newer.*

There are three ways of defining transformation:

Simple definition:

```
{
    "columns":
    {
        "target_column_name": "source_column_name-><type>"
    }
}
```

Object definition:

```
{
    "columns":
    {
        "target_column_name":
        {
            "name": "source_column_name",
            "transform": "<type>"
        }
    }
}
```

Full object definition

```
{
    "columns":
    {
        "target_column_name":
        {
            "name": "source_column_name",
            "transform":
            {
                "type": "<type>"
            }
        }
    }
}
```

Where `<type>` is one of the supported value: `Lowercase`, `Uppercase`, `IsNullOrEmpty`, `IsNotNullOrEmpty` or `Length`. Transformation names are case-insensitive. All three ways of defining a transformation are functionally and technically equal. We recommend using the simplified string notation, as it is the least verbose of all three.

Example

Given the following table **SourceTable:**

Name	Surname	Age
Marcin	Otorowski	34
John	*<null>*	51
Kate	Binks	25

And the following JSON step definition:

```
{
  "id": 1,
  "type": "map",
  "name": "Example of transform-columns",
  "source": "SourceTable",
  "columns": {
    "Name": "Name",
    "Name_Uppercase": "Name->uppercase",
    "Surname_IsProvided": "Surname->isnotnullorempty",
    "Surname_Length": "Surname->length"
  },
  "target": "TargetTable_TransformedColumns"
}
```

The following result table **TargetTable_TransformedColumns** is expected:

Name	Name_Uppercase	Surname_IsProvided	Surname_Length
Marcin	MARCIN	1 *(bit)*	9
John	JOHN	0 *(bit)*	0
Kate	KATE	1 *(bit)*	5

Arbitrary types, casting and converting

Using the `type` property lets you define a custom cast or custom type length. This common property is available for all column types, see more information in chapter Arbitrary column types.

Fallback values

It is possible to define a custom value returned, if the source column does not exist in the table. For more information about the usage of fallback values, see the Fallback columns chapter.

Regular expression match

Note:
This column mapping type is available in ETL module version 1.1.271 and newer.

The value can be checked against a Regular Expression mask. This mapping returns `1` (true) if the given column matches the expression, or `0` (false) if it does not match the expression.

To define a Regular Expression match, an object notation in the following format is required:

```
{
    "columns": {
        "target_column_name":
        {
            "column": "<name-of-source-column>",
            "regex": "<regular-expression>"
        },
        [...]
    }
}
```

Where `<regex>` is a valid Regular Expression pattern. There are no more properties required or defined by regex-columns.

Note:
More information about Regular Expressions and valid patterns can be found in internet. The following website is a great resource for both basic and advanced concepts:
https://www.regular-expressions.info/

Special characters and escaping sequences in Regular Expression patterns

Regular expressions use \ (backslash) as an escape sequence. Bear in mind the same escape sequence is used by JSON schema. This means, that when escaping a regular expression inside a JSON file, it is necessary to perform "double-escaping".

For example, to test whether the name consists of letters, followed by a dot, followed by numbers, the following Regular Expression can be used:

```
^[a-zA-Z]+\.[0-9]+$
```

Explanation:

- ^ means this is the beginning of the string
- $ means the string ends here
- [a-zA-Z]+ - one or more instances of characters from range a-z or A-Z.
- [0-9]+ - one or more digit
- \. - literal dot. Dot has a special meaning in Regular Expressions (any character), and to use it literally it must be escaped with \ (backslash).

However, the backslash in the regular expression must be escaped once more when used in JSON file, so that it is not interpreted as escape JSON sequence. The following JSON would be valid:

```
"regex": "^[a-zA-Z]+\\.[0-9]+$"
```

Example

Given the following table **SourceTable:**

Name	Mail
Marcin	marcin@raynet.de
John	john@raynet.ch

Name	Mail
Kate	kate@contoso.com

And the following JSON step definition:

```
    {
      "id": 1,
      "type": "map",
      "name": "Example of auto-columns",
      "source": "SourceTable",
      "columns": {
        "Name": "Name",
        "WorksInRaynet": {
          "column": "Mail",
          "regex": "@raynet\\.(de|ch|com)$" // this will check if the Mail column ends
with @raynet.de, or @raynet.ch or @raynet.com.
        }
      },
      "target": "TargetTable_Regex"
    }
```

The following result table **TargetTable_Regex** is expected:

Name	WorksInRaynet
Marcin	1
John	1
Kate	0

Fallback values

It is possible to define a custom value returned, if the source column does not exist in the table. For more information about the usage of fallback values, see the Fallback columns chapter.

Switch-case statement

Swich-case statement can be used to define branches for possible values.

The definition has the following syntax:

```
"<target-column-name>":
{
    "column": "<source-column-name>",
    "switch": [
```

```
        {
          "case": "<case-1>",
          "then": "<value-1>"
        },
        {
          "case": "<case-2>",
          "then": "<value-2>"
        },
        {
          "default": "<default-value>"
        }
      ]
    }
}
```

It is not possible to mix the property `default` with `case`/`then` in the same object. The default section is optional, and number of case-then tokens can be any. The default value must be always the last in the collection.

Example

Given the following table **SourceTable:**

Name	Country
John	US
Helmut	DE
Marcin	PL
Sergey	UA
Sami	FI
Alon	UK

And the following JSON step definition:

```
{
  "id": 1,
  "name": "Example of switch-case",
  "type": "map",
  "source": "Countries",
  "target": "Countries_SwitchCase",
  "columns": {
    "Name": "Name",
    "Country": {
      "column": "Country",
      "switch": [
```

```
            { "case": "PL", "then": "Poland" },
            { "case": "DE", "then": "Germany" },
            { "case": "US", "then": "United States" },
            { "case": "FI", "then": "Finland" },
            { "case": "UA", "then": "Ukraine" },
            { "default": "Other country" }
          ]
        }
      }
    }
```

The following result table **Countries_SwitchCase** is expected:

Name	Country
John	United States
Helmut	Germany
Marcin	Poland
Sergey	Ukraine
Sami	Finland
Alon	Other country

Fallback values

It is possible to define a custom value returned, if the source column does not exist in the table. For more information about the usage of fallback values, see the Fallback columns chapter.

Custom SQL statement

In some cases, extra customization and flexibility may be required, which is beyond of scope of simple transforms or switch-case statements. If this happens, you can use custom SQL snippets for columns.

You can use all language construct of SQLite, like string, date and other operations. Additionally, some extra functions implemented by ETL library are available (the list and reference can be found in chapter Extra functions)

```
{
    "columns":
    {
        "source_column": "source_column",
        "source_column_normalized":
        {
```

```
            "sql": "SUBSTR(source_column, 1, 3)"
        }
    }
}
```

The list of standard functions can be found here: https://sqlite.org/lang_corefunc.html

Example

Given the following table **SourceTable:**

Name	Surname	Age
Marcin	Otorowski	34
John	Smith	51
Kate	Binks	25

And the following JSON step definition:

```
    {
      "id": 1,
      "type": "map",
      "name": "Example of custom SQL columns",
      "source": "SourceTable",
      "columns": {
        "Name": "Name",
        "IsJohn": { "sql": "iif([Name] = \"John\", \"Hello John\", \"Hello, do I know
you?\")" },
        "Substring": { "sql": "SUBSTR([Name], 1, 3)" },
        "Random": { "sql": "random()" }
      },
      "target": "TargetTable_SqlColumns"
    }
```

The following result table **TargetTable_SqlColumns** is expected:

Name	IsJohn	Substring	Random
Marcin	Hello John	Joh	-6849993107088852193
John	Hello, do I know you?	Mar	5566813575801383258

Name	IsJohn	Substring	Random
Kate	Hello, do I know you?	Kat	585046792565859908 8

Note:
Special character " and \ should be escaped in JSON file. This is why in the example above, a \" token is used instead of a single ".

Arbitrary types, casting and converting

Using the `type` property lets you define a custom cast or custom type length. This common property is available for all column types, see more information in chapter Arbitrary column types.

Advanced topics

This chapter describes the usage of advanced features of the mapping function.

- Fallback columns
  Fallback is a value, column, SQL statement or any valid column mapping which is used, if the original source does not contain the required column.
- Union mapping
  Mapping can use more than one table as a source, in which case it performs a concatenation of two or more tables.
- Arbitrary column types
  Database column types are inferred from usage. It is possible to customize this behavior on column-basis, by changing the required type or its length / range.
- Inferring remaining columns
  Columns do not have to be all referred by their names. If you want to map them all (or all which have not been referenced explicitly) you can force the ETL engine to retrieve them automatically.

Fallback columns

Note:
This column mapping type is available in ETL module version 1.1.300 and newer.

ETL Library makes few assumptions about the existence of tables or columns. Without any overrides, by default all tables are optional (do not have to exist at all). In case of columns, the optional state looks a bit different though:

- All columns not specifically named, but referenced via the usage of `mapRemaining` attribute are optional.
- Named columns must exist.

It is possible to define a fallback for every column that references a single column name. Fallback is another expression, which is taken if the original source column does not exist.

To define a fallback, set-up the fallback property for each affected column, where the value of fallback follows the same syntax as a typical column definition (minus column type and fallback). For example, to define that a column `Name` should be taken, and if it does not exist a value of "`n/a`" is expected, use the following:

```
"UserName": {
  "column": "Name",
  "fallback": {
     "value": "n/a"
   }
}
```

Fallback is supported for the following column mappings:

- Simple column mapping
- Transformed column mapping
- Regular expression mapping
- Switch-case mapping

Tips:

- You can use a fixed value with null target value to denote that a null value must be placed for a non-existing column.
- You can use a typical mapping syntax to use value from another column
- It is not possible to define a different type for fallback column. Fallback always uses the type from the column it belongs to.
- Additionally, it is not possible to nest a fallback in another fallback.

Example

Given the following table **Table1:**

Name	Country	Age
Helmut	DE	20

Name	Country	Age
Marcin	PL	40

With the following JSON file

```
{
  "id": 1,
  "type": "map",
  "name": "Fallback Demo",
  "source": "Table1",
  "columns": {
    "Name": "Name",
    "City": {
      "column": "City",
      "fallback": {
        "value": "n/a"
      }
    },
    "BornYear": {
      "column": "Year",
      "fallback": {
        "sql": "2021 - Age"
      }
    }
  },
  "target": "Table2"
}
```

The following **Table2** is produced:

Name	City	BornYear
Helmut	n/a	2001
Marcin	n/a	1981

Union mapping

It is possible to union two or more tables, regardless of whether the number and names of columns are the same. To define more than one table during the mapping, simply use an array as a value of the `source` property, for example:

```
{
  "id": 1,
  "type": "map",
  "name": "Example of union",
  "source": ["SourceTable1", "SourceTable2"],
  "columns": {
    "Name_Uppercase": "Name"
```

```
    },
    "target": "TargetTable_TransformedColumns",
    "mapRemaining": true
}
```

You can also use wildcards (for example `SourceTable*`) to match more than one table (more information about wildcards in the following chapter: Wildcard joining).

The result table will have the same number of rows as all involved tables combined. The number of columns in the target table will be the same as all columns in all tables combined, minus duplicates.

Example

Given the following table **Table1:**

Name	Country	Age
Helmut	DE	20
Marcin	PL	40

and the following table **Table2:**

Name	Residence	Age
Helmut	DE	30
Alon	UK	50

With the following JSON file

```
{
  "id": 1,
  "type": "map",
  "name": "Union two tables",
  "source": ["Table1", "Table2"],
  "columns": {
    "Name": "Name",
    "Country": "Country",
    "Residence": "Residence",
    "Age": "Age"
  },
  "target": "Table3",
  "mapRemaining": true
}
```

The following **Table3** is produced (note that there is no de-duplication, and extra columns are

added for rows which were not had them before, thus producing `null` values in columns `Country` and `Residence`).

Name	Country	Residence	Age
Helmut	DE	*<null>*	20
Marcin	PL	*<null>*	40
Helmut	*<null>*	DE	30
Alon	*<null>*	UK	50

Arbitrary column types

By default, the type of each column is inferred from the context. For simply queries, the type may be inferred from the source table, while for more complex it is either based on the type of the expression or the actual value type.

In some cases, it may be required to use a specific type, for example to enforce that a string-based column is written as CHAR(20) and not NVARCHAR(MAX). The same applies to numeric formats, which also may have different precisions (`INT`, `SMALLINT`, `TINYINT` etc.).

To define an arbitrary type for a column, use the property *type*, for example:

```
{
  "id": 1,
  "type": "map",
  "name": "Example for arbitrary column types",
  "source": "SourceTable",
  "columns": {
    "TypeFromParent": { "from": "ColumnA" },
    "TypeFromExpression": { "from": "ColumnA", "transform": "Uppercase" },
    "ArbitraryType": { "from": "ColumnA", "type": "nvarchar(100)" },
  },
  "target": "TargetTable"
}
```

This property is optional - when omitted, the type will be inferred automatically. Note that the type can be defined for any type of column: simple mapping, a fixed value, custom SQL etc.

Supported types

Several types and their aliases are accepted. You can use MSSQL types or DbType enum values (https://docs.microsoft.com/en-us/dotnet/api/system.data.dbtype?view=net-5.0). Depending on the implementation, some types may be unavailable (for example XML or currency).

Examples of valid types (the list is not comprehensive, see the referenced link for a full list):

- `int`
- `bigint`
- `smallint`
- `nvarchar`

- varchar(max)
- char(20)
- datetime
- numeric(18,0)
- string
- bit

Inferring remaining columns

Map step requires, that you define the list of columns to be mapped. If you do not know the exact names ahead-of-time, or if just a few columns need to be transformed and the others should be taken as they are, it is possible to use the automatic inferring.

To do this, set the value of optional parameter *mapRemaining* on the step level to *true*, for example:

```
{
  "id": 1,
  "name": "Map with remaining",
  "columns": {
    "Column1": "source1",
    "Column2": "source2"
  },
  "target": "OutputTable",
  "source": "InputTable",
  "mapRemaining":  true
}
```

Example

Let's assume that the input table **InputTable** has the following structure:

col1	col2	col3
Marcin	Otorowski	Renault
John	Smith	Chrysler
Kate	Binks	Peugeot

When the ETL process is started for the sample JSON file, the following table **OutputTable** is returned:

Column1	Column2	col3
Marcin	Otorowski	Renault
John	Smith	Chrysler

Column1	Column2	col3
Kate	Binks	Peugeot

If the parameter *mapRemaining* was set to false or omitted, the following would be produced:

Column1	Column2
Marcin	Otorowski
John	Smith
Kate	Binks

Note that the *col3* was not defined in JSON, and yet it made to the *OutputTable* due to the switch *mapRemaining* set to *true*. On the other hand, columns *col1* and *col2* are not present anymore, because the engine has detected they were referenced by the standard mapping, and thus are potentially not needed anymore. If you want to include them, you must add a manual definition for them.

Once *mapRemaining* is set to *true*, the ETL engine will automatically restore all columns with the following exceptions:

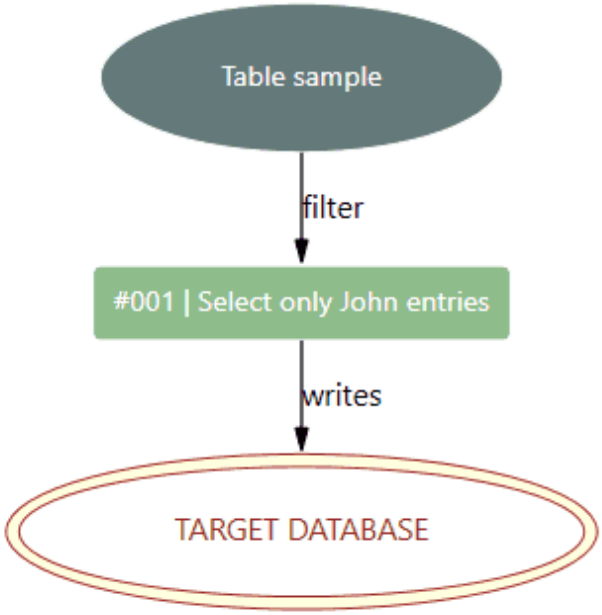
- Columns referenced as source columns for simple mappings (see simple mappings).
- Columns referenced as source columns for transformation mappings (see transforming of values).
- Columns referenced as one of the sources for aggregated mapping (see columns aggregation).

Columns referenced by custom SQL mappings are not taken into account when evaluating the *mapRemaining* switch.

Filtering

Filter step takes one value as input and produces another table with the same schema, but with the same or lower number of rows, based on filter conditions (subtraction).

The simplest definition has the following form:



Select only John entries

ID
001

Input
Table sample

Conditions
Name = John

Output
Table sample_filtered

```
{
    "steps":
    [
        {
            "id": 1,
            "name": "Select only John entries",
            "target": "sample",
            "source": "sample_filtered",
            "type": "filter",
            "conditions":
            [
                {
                    "Name": "John"
                }
            ]
        }
    ]
}
```

Source is by default optional, meaning that the table does not have to be existing. If the table is missing, the step will not be executed and its target table will not be written. There is a way to define that the source is required, in which case in case of a missing source table the step will fail and report an error. More information about setting up required steps can be found in the following chapter: Optional and required tables.

Example

Given the following table **Table_sample**:

Name	Value
Marcin	123
John	456
Josh	abc
John	aaa

And the following JSON:

```
{
  "id": 1,
  "type": "filter",
  "name": "Example of filtering",
  "source": "Table_sample",
  "conditions": [
    {
      "Name": "John"
    }
  ],
  "target": "TargetTable_Filtered"
}
```

The following result table **TargetTable_Filtered** is expected:

Name	Value
John	456
John	aaa

Logical operators

Both AND and OR are possible, with a Mongo-inspired syntax. To define AND condition, use $and as the name, and array of sub-conditions as a value:

```
"conditions":
[
    {
        "$and":
        [
            {
                "Name": "John"
            },
            {
                "Value": "abc"
            }
        ]
    }
]
```

The equivalent operator for OR operation is `$or`. Nesting is also possible to build some complex scenarios.

Comparison and other operators

The following operators are supported:

Operator	Description and usage
`$gt`	The value is greater than
`$gte`	The value is greater or equal than
`$lt`	The value is less than
`$lte`	The value is less or equal than
`$ne`	The value is not equal to
`$eq`	The value is equal than
`$contains`	The value contains a string
`$startsWith`	The value starts with
`$endsWith`	The value ends with

The syntax for defining this operators is also shared with Mongo. You define the name of the column as a key, the value is another object with your operator, and its value is the value you want to use. A few examples with explanation:

Name is not equal to Marcin:

```
{
    "Name":
    {
        "$ne": "Marcin"
    }
}
```

Age is greater than 17

```
{
    "Age":
    {
        "$gt": 17
```

```
    }
}
```

CompanyName contains Raynet

```
{
    "CompanyName":
    {
        "$contains": "Raynet"
    }
}
```

CompanyName does not contain Raynet (negation of contains)

```
{
    "CompanyName":
    {
        "$not":
        {
            "$contains": "Raynet"
        }
    }
}
```

Custom filtering

> **Note:**
> This is advanced topic

Any conditions built using the standard operator are converted to an expression tree which is queried against the sandbox database. For even greater flexibility, SQLite language constructs can be used.

To use SQL, simple use the following operator:

Operator	Description and usage
`$sql`	The expression to evaluate. It should return True or False.

An example of a filter would be:

```
[...]
"conditions": [
    {
        "$and": [
            {
                "PrimaryUser": { "$ne": "DGFF" }
            },
            {
                "$sql": "DeviceManufacturer LIKE '%HPE%'"
            }
        ]
```

```
    }
]
[...]
```

This condition checks if both PrimaryUser is not equal to DGFF (standard operator) AND that the value of column DeviceManufacturer contains HPE. Also note that you can freely combine built-in expression with SQL expression as we did in this example.

Grouping and deduplicating

Grouping is a process which covers one of the following use cases:

- Having a single table, it writes the output table that has the exact amount of rows, but with extra meta-columns containing a "group identifier". Two or more elements considered to be the same have always the same value in the group column. The second extra column being added is the count of rows in the given group.
- Having a single table, it writes the output table that contains aggregated (de-duplicated / distinct) rows, where the values in each non-grouped column is aggregated with a specified function (maximum, minimum, average, concatenated value, coalesced value). An extra column with number of grouped rows is added.

The following snippet shows a functional example required to group rows and enter the information about the group:

```
{
  "id": 1,
  "type": "group",
  "name": "Example of grouping (deduplicate)",
  "source": "Users",
  "by": [ "Name", "E-Mail" ],
  "target": "Users_Grouped_Deduplicated"
}
```

You can opt-in for de-duplication by using the "action" object:

```
{
      "id": 2,
      "type": "group",
      "name": "Example of grouping (deduplicate)",
      "source": "Users",
      "by": [ "Name", "E-Mail" ],
      "target": "Users_Grouped_Deduplicated",
      "action": {
        "type": "deduplicate"
      }
}
```

At minimum the following properties are required:

- ID (must be unique)
- Type (must be set to "*group*")
- Source (must be one of the following):

- A string representing the table name
- An integer representing the source as another step
- An object with property "table" set to the name of the source table
- An object with property "step" set to the ID of the source step

Source is by default optional, meaning that the table does not have to be existing. If the table is missing, the step will not be executed and its target table will not be written. There is a way to define that the source is required, in which case in case of a missing source table the step will fail and report an error. More information about setting up required steps can be found in the following chapter: Optional and required tables.

Following parameters are optional:

- **by**
  This defines a single column or a list of columns to be used as grouping keys. Equivalent to SQL statement `Select * FROM <table> GROUP BY <columns>.`
  If you omit this parameter or set it to an empty list, then all columns are used for grouping.
- **target**
  If omitted, the output table is temporary and does not get saved during the LOAD phase.
- **action**
  The action to execute. This can be one of the following:
  - If the value is a string, then the action of a given type is used. Supported values are *deduplicate* and *recognize*. The values are case-insensitive.
  - If the value is an object, then its property ***type*** is used to determine the action. The value of parameter *type* should be either *deduplicate* or *recognize*. The values are case-insensitive. There are some extra properties available when using the object syntax.
  - If the action is omitted, the *recognize* action is used as the default.

Recognize action

This type of action is used to group and recognize grouped values, but without removing any rows yet. Instead, the ETL engine compares the values in given columns, groups matching rows and for each row writes additional two columns:

- **GROUP_KEY**: Containing the locally-unique identifier of the group (integer-based, starting from 1)
- **GROUP_COUNT**: Containing the number of rows within the same group.

You can customize the naming of both properties by setting the properties *groupKeyColumn* and *groupCountColumn* respectively.

Deduplicate action

This type of action is used to group and consolidate grouped rows. The non-grouped columns are aggregated using a specific algorithm. ETL engine writes additional extra column for each row:

- **GROUP_COUNT**: Containing the number of rows within the same group.

The default setting of this action use coalescing to produce aggregated value (getting the first non-null value). You can customize the behavior of aggregation by setting the object **columns**, where key is the name of the column to aggregate, and the value is the aggregation method. The syntax for aggregation is similar to the definition for the join step.

Example #1

Given the following table **Users**

Name	LastSeen	Device	E-Mail
Marcin	2020-01-02 00:00:00.000	Android	marcin@contoso.com
Marcin	2020-01-03 00:00:00.000	Windows	marcin@contoso.com
Marcin	2020-01-01 00:00:00.000	Apple	*<null>*
Denis	2020-03-03 00:00:20.000	Windows	denis@contoso.com
Denis	2021-02-02 00:10:00.000	Android	*<null>*
Andreas	2021-02-02 00:20:00.000	Windows	andreas@contoso.com

and the following JSON file:

```
{
  "id": 1,
  "type": "group",
  "name": "Example of grouping (only marking)",
  "source": "Users",
  "by": [ "Name", "E-Mail" ],
  "target": "Users_Grouped_Mark",
  "action": {
    "type": "recognize"
  }
```

```
},
{
  "id": 2,
  "type": "group",
  "name": "Example of grouping (deduplicate)",
  "source": "Users",
  "by": [ "Name", "E-Mail" ],
  "target": "Users_Grouped_Deduplicated",
  "action": {
    "type": "deduplicate"
  }
}
```

The following two tables are created:

Table **Users_Grouped_Mark**:

Name	E-Mail	LastSeen	Device	GROUP_KEY	GROUP_COUNT
Marcin	marcin@contoso.com	2020-01-02 00:00:00.000	Android	5	2
Marcin	marcin@contoso.com	2020-01-03 00:00:00.000	Windows	5	2
Marcin	*<null>*	2020-01-01 00:00:00.000	Apple	4	1
Denis	denis@contoso.com	2020-03-03 00:00:20.000	Windows	3	1
Denis	*<null>*	2021-02-02 00:10:00.000	Android	2	1
Andreas	andreas@contoso.com	2021-02-02 00:20:00.000	Windows	1	1

The other table **Users_Grouped_Deduplicated** has the following rows:

Name	E-Mail	LastSeen	Device	GROUP_COUNT
Marcin	marcin@contoso.com	2020-01-02 00:00:00.000	Android	2
Marcin	NULL	2020-01-03 00:00:00.000	Windows	1
Denis	*<null>*	2020-01-01 00:00:00.000	Apple	1

Name	E-Mail	LastSeen	Device	GROUP_COUNT
Denis	denis@contoso.com	2020-03-03 00:00:20.000	Windows	1
Andreas	andreas@contoso.com	2021-02-02 00:20:00.000	Windows	1

Example #2

With the same input **Users** as in the first example and the following JSON (yellow marks the difference between Example #1 and Example #2):

```
    {
      "id": 3,
      "type": "group",
      "name": "Example of grouping (only marking)",
      "source": "Users",
      "by": [ "Name", "E-Mail" ],
      "target": "Users_Grouped_Mark2",
      "action": {
        "type": "recognize",
        "groupKeyColumn": "ETL-GROUP",
        "groupCountColumn": "ETL-COUNT"
      }
    },
    {
      "id": 4,
      "type": "group",
      "name": "Example of grouping (deduplicate)",
      "source": "Users",
      "by": [ "Name", "E-Mail" ],
      "target": "Users_Grouped_Deduplicated2",
      "action": {
        "type": "deduplicate",
        "groupKeyColumn": "ETL-GROUP",
        "groupCountColumn": "ETL-COUNT",
        "columns": {
          "Device": "concat",
          "LastSeen": "max"
        }
      }
    }
```

The following table are created:

Table **Users_Grouped_Mark2**:

Name	E-Mail	LastSeen	Device	ETL-GROUP	ETL-COUNT
Marcin	marcin@contoso.com	2020-01-02 00:00:00.000	Android	5	2
Marcin	marcin@contoso.com	2020-01-03 00:00:00.000	Windows	5	2
Marcin	*<null>*	2020-01-01 00:00:00.000	Apple	4	1
Denis	denis@contoso.com	2020-03-03 00:00:20.000	Windows	3	1
Denis	*<null>*	2021-02-02 00:10:00.000	Android	2	1
Andreas	andreas@contoso.com	2021-02-02 00:20:00.000	Windows	1	1

The other table **Users_Grouped_Deduplicated2** has the following rows:

Name	E-Mail	LastSeen	Device	ETL-COUNT
Marcin	marcin@contoso.com	2020-01-03 00:00:00.000	Android | Windows	2
Marcin	NULL	2020-01-03 00:00:00.000	Windows	1
Denis	*<null>*	2020-01-01 00:00:00.000	Apple	1
Denis	denis@contoso.com	2020-03-03 00:00:20.000	Windows	1
Andreas	andreas@contoso.com	2021-02-02 00:20:00.000	Windows	1

Note the differences between both examples:

- For the first entry, without specifying any aggregation, the first non-null empty was taken, which resulted in **Android** in the first example. In the second example, the column is built using **Concat**, which means that the value in the Device column joins all grouped values (with | as a separator).
- In the same row, the last seen date is different. In the first example **2020-01-02 00:00:00.000** was taken but in the second one where the **MAX** aggregation was used, the date is set to **2020-**

01-03 00:00:00.000 (the maximum of two values in the group).

- The extra columns **ETL-COUNT and ETL-GROUP** are named after the values that were defined in the second example, but omitted from the first one. In the first one, their names are defaulted to **GROUP_KEY** and **GROUP_COUNT**.

Default aggregation of duplicated values

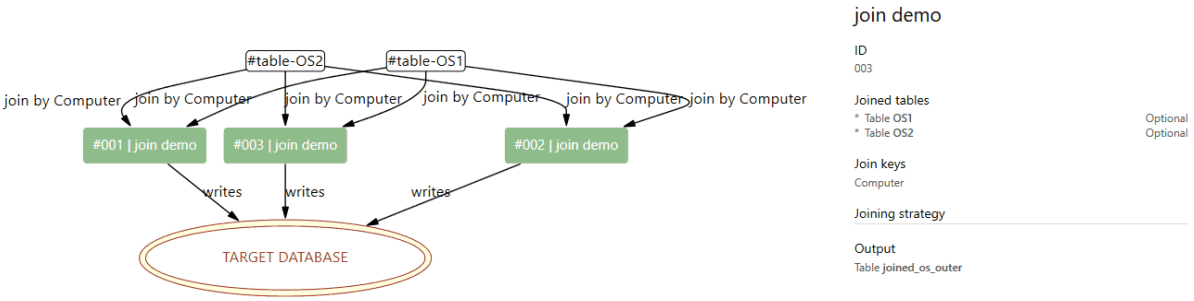
You can use JSON property `defaultAggregation` on the action object to define the required aggregation, should there be no other defined on a column-base.

Joining

Joining is an operation which takes two or more tables and combines them to a single on. Combination of rows is performed using a special column or set of columns, called joining keys. They should be locally unique to make actual sense.

Since two tables may but do not have to have all the same rows and keys, conflicts may arise:

- Left table may have some unmatched rows in the right table
- The right table may have some unmatched rows in the left table



There are three basic strategies which dictate what to do in this case:

- **Outer joining**
  Will combine all matching rows, and for all unmatched it will take them as they-are, using NULL values are default for unmatched values. Typical use case: list of devices scanned by system A and system B may have a lot of entries in common, but some devices may be present only in system A and some devices only in system B. Outer join will ensure that all devices are present in the target table, even unmatched devices from system A and unmatched devices from system B.

- **Left joining**
  Will treat the left column as a master table and write all rows which were matched between left and right + any unmatched rows from the left table. Unmatched rows from the right column will be discarded. A typical example is lookup, in which the left table is the list of customers and the right table is the list of addresses. In the result

table, we are interested in all results from the left table (customers) with as much extra information as possible from the address table, but we do not want to include unmatched addresses.

- **Inner joining**
  Will write only rows which exist in both tables. Unmatched rows from the left table and the right table will be ignored. Typical example is where the left table contains names and the right tables surnames. The output table should have name and the surname, we do not want partial matches with only name or only surname.

Join step requires a bit of extra information about the key used to join (ID). This key must be present in all joined tables, otherwise an exception will be thrown.

You may also join more tables at once (within the same strategy).

Joined tables are by default optional, meaning that the tables do not have to be existing. If any table is missing it will not be joined. If only a single table exists, then it is taken as-is and returned as the output. If all tables are missing then the step will not be executed and its target table will not be written. There is a way to define required tables, in which case in case of a missing source table the step will fail and report an error. More information about setting up required steps can be found in the following chapter: Optional and required tables.

Cell merging

Join operation uses a convention based approach. When defining a join, the only information required is:

- Which tables need to be joined
- Which columns are the key used to join them

All tables taking a part in the join must have all columns specified as the keys. For other columns, the following logic is applied:

- If a column exist in exactly one of joined tables, it is taken as-is
- If a column exists in two or more tables, the values in it are merged using a specific aggregation strategy. The default aggregation is to join the values with a pipe " | " separator. This may not always be the best choice, so other strategies are also available.

For example, given the following:

Table **OS1**

Computer	OperatingSystem	ValueFromOS1
1	Windows	A
2	Linux	B

Computer	OperatingSystem	ValueFromOS1
3	macOS	C
4	Android	D

Table **OS2**

Computer	OperatingSystem	ValueFromOS2
2	BlackBerryOS	AA
3	Ubuntu	BB
4	Windows	CC
5	Windows	DD

and the following JSON definition

```
{
  "steps": [
    {
      "id": 1,
      "type": "join",
      "name": "join demo",
      "sources": [ "OS1", "OS2" ],
      "on": [ "Computer" ],
      "strategy": "left",
      "target": "joined_os_left"
    },
    {
      "id": 2,
      "type": "join",
      "name": "join demo",
      "sources": [ "OS1", "OS2" ],
      "on": [ "Computer" ],
      "strategy": "inner",
      "target": "joined_os_inner"
    },
    {
      "id": 3,
      "type": "join",
      "name": "join demo",
      "sources": [ "OS1", "OS2" ],
      "on": [ "Computer" ],
      "strategy": "outer",
      "target": "joined_os_outer"
    }
  ]
}
```

The following tables are returned:

Table **joined_os_left**

Computer	OperatingSystem	ValueFromOS1	ValueFromOS2
1	Windows	A	*<null>*
2	Linux	B	AA
3	macOS	C	BB
4	Android	D	CC

Table **joined_os_inner**

Computer	OperatingSystem	ValueFromOS1	ValueFromOS2
2	Linux	B	AA
3	macOS	C	BB
4	Android	D	CC

Table **joined_os_outer**

Computer	OperatingSystem	ValueFromOS1	ValueFromOS2
1	Windows	A	*<null>*
2	Linux	B	AA
3	macOS	C	BB
4	Android	D	CC
5	Windows	*<null>*	DD

Remarks and points of interest:

- Table **joined_os_outer** has most of rows, as it contains all unique values from the column **Computer** present in both tables.

- Table **joined_os_inner** has the least of rows, because it contains only values present in both tables.
- No extra definition for "overlapping" columns was present in the JSON file, which means that the default COALESCE (first non-null) aggregation will be used. This is why the column **OperatingSystem** in all three output tables contains a single non-empty value. The order of coalescing a value should be inferred from the order of joined tables.
- Columns **ValueFromOS1** and **ValueFromOS2** are unique in all tables, and are taken as-is. Missing values are filled with *<null>*.

Overriding cell merging strategy

As already mentioned, the default strategy of merging/aggregation of overlapping columns uses concatenation with a pipe (|) character.

There are two ways of overriding the strategy:

- For all columns in a step
- Specific types for selected columns.

The order how ETL resolves defined aggregations is:

1. Column-specific aggregation
2. Step-scoped aggregation
3. Default aggregation (concatenate values using pipe as separatator).

Step merging strategy

Note:
This aggregation is available in ETL module version 1.1.281 and newer.

The default aggregation used for all columns in the given steps can be defined by defining the value of a property called `defaultAggregation`. Below is a sample code to demonstrate:

```
{
    "steps": [
        {
            "id": 1,
            "type": "join",
            "name": "join demo",
            "sources": ["OS1", "OS2"],
            "on": ["Computer"],
            "strategy": "outer",
            "target": "joined_os_outer_max",
            "defaultAggregation": "coalesce"
        },
    ]
}
```

The value can be a string or an object. For more information about supported aggregations, see the following chapter: Defining aggregations.

Column-specific merging strategy

It is possible to specify custom aggregation (similar to what the complex column mapping does). In order to do that, specify a special section which will define how to handle conflicts. The value of that property (`conflicts`) is an object where each property is the name of the column to define, and the value is the aggregation method. The value can be either a string or an object. For more information about supported aggregations and the ways of resolving conflicts, see the following chapter: Defining aggregations.

The sample below shows how to get the max value from the **OperatingSystem** column (1), how to concatenate all values using custom separator (2) and how to coalesce columns by selecting the first non-empty value (3):

```
{
    "steps": [
        {
            "id": 1,
            "type": "join",
            "name": "join demo",
            "sources": ["OS1", "OS2"],
            "on": ["Computer"],
            "strategy": "outer",
            "target": "joined_os_outer_max",
            "conflicts": {
                "OperatingSystem": "max"
            }
        },
        {
            "id": 2,
            "type": "join",
            "name": "join demo",
            "sources": ["OS1", "OS2"],
            "on": ["Computer"],
            "strategy": "outer",
            "target": "joined_os_outer_concat",
            "conflicts":
            {
                "OperatingSystem": {
                    "type": "concat",
                    "separator": " or "
                }
            }
        },
        {
            "id": 3,
            "type": "join",
            "name": "join demo",
            "sources": ["OS1", "OS2"],
            "on": ["Computer"],
            "strategy": "outer",
            "target": "joined_os_outer_coalesce",
            "conflicts":
            {
                "OperatingSystem": "coalesce"
```

```
                }
            }
        ]
}
```

Like already mentioned, the way of resolving a conflict is defined via a mapping of a column name and a matching value (string or an object) representing the aggregation. In the example above, both syntax variants are shown: for column step 2, a concatenation with a custom separator is used (object notation), Steps 1 and 3 use a simple string notation.

Using the same input tables as in the chapter Cell merging, the following tables are written with the above JSON (the yellow color highlights important changes and differences):

Table **joined_os_outer_max**

Computer	OperatingSystem	ValueFromOS1	ValueFromOS2
1	Windows	A	*<null>*
2	Linux	B	AA
3	Ubuntu	C	BB
4	Windows	D	CC
5	Windows	*<null>*	DD

Table **joined_os_outer_concat**

Computer	OperatingSystem	ValueFromOS1	ValueFromOS2
1	Windows	A	*<null>*
2	Linux or BlackBerryOS	B	AA
3	macOS or Ubuntu	C	BB
4	Android or Windows	D	CC
5	Windows	*<null>*	DD

Table **joined_os_outer_coalesce**

Computer	OperatingSystem	ValueFromOS1	ValueFromOS2
1	Windows	A	*<null>*
2	Linux	B	AA
3	macOS	C	BB
4	Android	D	CC
5	Windows	*<null>*	DD

Remarks and points of interest:

- Table **joined_os_outer** has most of rows, as it contains all unique values from the column **Computer** present in both tables.
- Table **joined_os_inner** has the least of rows, because it contains only values present in both tables.
- No extra definition for "overlapping" columns was present in the JSON file, which means that the default coalesce (first non-null) will be used. This is why the column **OperatingSystem** in all three output tables contains a single non-empty values from tables ordered by the master joining order.
- Columns **ValueFromOS1** and **ValueFromOS2** are unique in all tables, and are taken as-is. Missing values are filled with *<null>*.

Multiple conflict handling strategies

You can define many conflict resolving strategies for each columns of your interest. The columns that you don't define will use the standard joining strategy (concatenation with pipe character).

```
        {
            "id": 3,
            "type": "join",
            "name": "join demo",
            "sources": ["OS1", "OS2"],
            "on": ["Computer"],
            "strategy": "outer",
            "target": "joined_os_outer_coalesce",
            "conflicts":
            {
                "OperatingSystem": "coalesce", // take the first not-empty operating
system
                "InventoryDate": "max" // take the maximum date
            }
        }
```

Priorities when joining the cell values

Unless otherwise specified, the columns will be aggregated in the order inferred from the list of tables (for example *OS1* then *OS2*, as in the JSON above). You can also define a custom ordering per-column - the topic is covered in the following chapter: Tables priority

Order of joining

Order of tables is important if (any of the following):

- LEFT join is used, or
- CONCAT aggregation is used for any column, or
- COALESCE aggregation is used

LEFT join is sensitive to the order, because the most left-table will be used as a join master, and switching the tables may produce different results in the output table.

CONCAT and COALESCE are also sensitive to the order because they process the value from left to right.

The joining order is determined by the order they appear in JSON file. You do not need to do anything extra if that order is the same for all conflicting columns in the joined tables. However, there may be a need to define a custom joining order on a column basis, which is described in the next chapter.

Note:

The tables matched by wildcard joining (for example *Table**) are ordered in an non-deterministic way if more than one table matches the wildcard.

Tables priority

For CONCAT and COALESCE columns, the default joining priority is used (the one that the user defined for the join step). In many cases, it may be required to change the priority on column basic. Consider the following example:

- Data from system A has generally all values, but a its column A is not 100% accurate (approximation). The column B in this table is always correct.
- Data from system B has less data, but a 100% correct value for that column A. It contains values in column B which on the other hand is unreliable.
- We need to combine two data sets in which
  - If the same row in both tables has a value in Table 1 Column A, this value should win over Table 2 Column B.
  - If the same row has only value of column A in Table 2, we should use it as a fallback.
  - If the same row in both tables has a value in Table 2 Column B, this value should win over

Table 1 Column B.

- If the same row has only value of column B in Table 1, we should use it as a fallback.

So essentially, we define the following priority:

- General joining priority: Table 1 > Table 2
- Priority for values in column A: Table 1 > Table 2
- Priority for values in column B: Table 2 > Table 1

To define this in JSON, use the following syntax:

```
{
    "id": 3,
    "type": "join",
    "name": "join demo",
    "sources": ["Table 1", "Table 2"],
    "on": ["Computer"],
    "strategy": "outer",
    "target": "joined_os_outer_coalesce",
    "conflicts":
    {
        "OperatingSystem": {
            "type": "coalesce",
            "priority": ["Table 2", "Table 1"]
        }
    }
}
```

The syntax for the priority is the same as the definition of source tables.

You do not have to define all tables in the *Priority* property. If you omit some of them, they will be treated as fallback anyway. For example, in the above snippet, if only Table 2 was provided in the priority, Table 1 and any other table will be always treated with a lower priority.

Note:

If you omit the tables you lose the control of defining the priority between omitted tables. In this case, the default priority will be used for them. The tables that you specified still win.

Wildcard priority

You can use wildcards when defining the joining priority. For example, if you generally prefer all tables containing a keyword "final" over "tables that end with "raw", you could do the following:

```
"conflicts":
{
    "OperatingSystem": {
        "type": "coalesce",
        "priority": ["*final*", "*raw"]
    }
}
```

You can also mix wildcard and non-wildcard strings. The priority order is not validated, should there be no column matching the priority order the next will be taken.

Example

Given the following tables containing approximated and actual age:

Table **Approximation**

Name	Age
Marcin	34
Ed	70

Table **Observation**

Name	Age
Marcin	35
Jeremy	20

And the following assumptions:

- Join two tables by the name
- Make sure that if the value from Observation and Approximation are in conflict, prefer the Observation as it is more accurate
- For entries present in only a single to these two tables, take the entry as-is

We would write the following JSON:

```
{
  "id": 1,
  "type": "join",
  "name": "Join with priority",
  "sources": [ "Approximation*", "Observation*" ],
  "strategy": "outer",
  "target": "Age",
  "on": "Name",
  "conflicts": {
    "Age": {
      "type": "coalesce",
      "priority": [ "*Obs*", "*Approx*" ]
    }
  }
}
```

Note: The example introduces a concept of wildcard joining, in which we do not have to specify the full name, but rather only a mask to match against. This works on both table selection level and on priority level.

The result would be

Table **Age**

Name	Age
Ed	70
Jeremy	20
Marcin	35

Remarks and points of interest:

- In the priority order `[ "*Obs*", "*Approx*" ]` we could also skip the last item (leaving only `[ "*Obs*" ]`) . Like mentioned before, columns which are not explicitly named in the priority list automatically receive the lowest priority.
- In the row *Marcin* the value of *Age* is 35. This is the value coming from the **Observation** table, although the master joining first took the **Approximation** table.
- Other values are taken from either of two tables, as they do not have the respective counter entries in the other table.

Wildcard joining

If your source tables are already well-formed and you do not want to define the joining tables manually, you can use so called wildcard syntax. If the name of the table contains an asterisk, all tables matching the pattern will be used for joining. For example, the previous step were OS1 and OS2 was joined together could be also represented with the following syntax:

```
{
    "id": 1,
    "type": "join",
    "name": "join demo",
    "sources": "Table*",
    "on": ["Computer"],
    "strategy": "outer",
    "target": "joined_os_outer_coalesce"
}
```

This is a relatively easy to way to make big joins of several tables, provided that:

- The match the pattern Table*
- They all contain a column “Computer”

If you have some prior knowledge about the columns and their content, and based on that you want to define preferences for joining, you can do it on the column basis:

```
{
    "id": 1,
    "type": "join",
    "name": "join demo",
    "sources": "Table*",
    "on": ["Computer"],
    "strategy": "outer",
    "target": "joined_os_outer_coalesce",
    "conflicts":
```

```
    {
        "OperatingSystem": {
            "type": "coalesce",
            "priority": ["Table 2", "Table 1"],
        }
    }
}
```

In this case, all tables matching the pattern will be joined, and in case of conflict in the column OperatingSystem the value from Table 2 will be preferred over Table 1 over some other, not mentioned tables. Wildcards for joining priority are also supported (see Tables priority for more information)

Column selection

If nothing else is provided, all columns from all joined tables will be present in the output table. It is possible to limit this by providing a white list of columns to be written. Any columns not mentioned in the list will be omitted from the target result (note: key columns are automatically included). To define the required columns, use the following syntax:

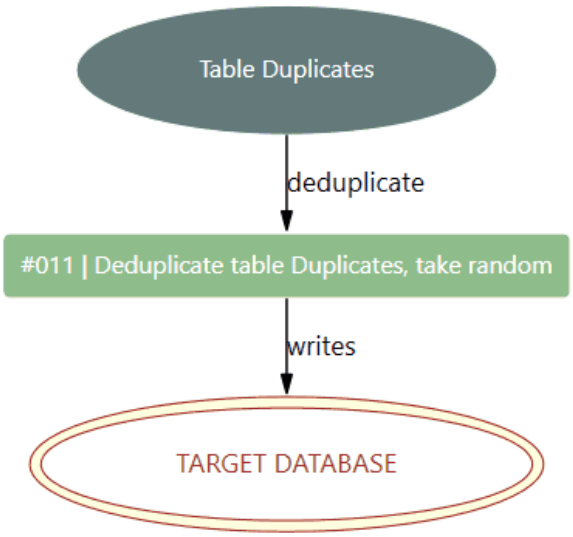
```
{
    "id": 1,
    "type": "join",
    "name": "join demo",
    "sources": ["Table1", "Table2", "Table3"],
    "columns": ["Name", "Surname", "Age"],
    "on": ["Computer"],
    "strategy": "outer",
    "target": "joined_os_outer_coalesce"
}
```

Advanced deduplicating

Deduplicating is a process of taking a table as an input, group the similar records by one or more columns, and then decide on each set how to proceed with the records.

Deduplication step has the following basic syntax:

```
{
    "id": 11,
    "name": "Deduplicate table Duplicates, take random",
    "type": "deduplicate",
    "source": "Duplicates",
    "target": "Normalized_Duplicates_Random",
    "by": [ "cn", "dn" ],
    "strategy": "random"
}
```



Deduplicate table Duplicates, take random

ID
011

Input
Table **Duplicates**

Group columns
cn, dn

Strategy
Take random row from conflicting rows

Output
Table **Normalized_Duplicates_Random**

In this sample, rows will be grouped by the same values in the cn and dn column. For each group, a random row will be picked and the rest will be skipped.

Selecting a random row may sound a bit weird, so the engine supports further options:

Strategy type	**Description**
None	If a duplicate is detected, all duplicated rows will be removed.
Random	Take random row (this is not reproducible between sessions).
Any	Take first row (this is reproducible between sessions).
All	Take all values (no deduplication)
MaxValue	Take a row with maximum value in a specified column (requires extra config, see below)
MinValue	Take a row with minimum value in a specified column (requires extra config, see below)

Source is by default optional, meaning that the table does not have to be existing. If the table is missing, the step will not be executed and its target table will not be written. There is a way to

define that the source is required, in which case in case of a missing source table the step will fail and report an error. More information about setting up required steps can be found in the following chapter: Optional and required tables.

Using MaxValue and MinValue strategy

These are handy built-in strategies that can be used with duplicated rows, for which one of their column values determines the priority (as a number, string, date etc.). Typical example would be: once two or more similar devices are found, take only the one that has been created latest, ignore the other devices).

Using of these strategies is only possible if a column name is provided, which can be done via the value parameter:

```
{
    "id": 11,
    "name": "Deduplicate table Duplicates, take random",
    "type": "deduplicate",
    "source": "Duplicates",
    "target": "Normalized_Duplicates_Random",
    "by": [ "cn", "dn" ],
    "strategy": {
        "name": "MaxValue",
        "value": "LastSeenDate"
    }
}
```

Splitting

Splitting is a process of taking an input table with N row, and producing N + X row (X >= 0) by splitting a value from a specific column.

Basic syntax for splitting:

```
{
  "id": 1,
  "name": "<name>",
  "type": "split",
  "source": "<source>",
  "target": "<target_if_persisted>",
  "column": "<the-column-to-split>"
},
```

Source is by default optional, meaning that the table does not have to be existing. If the table is missing, the step will not be executed and its target table will not be written. There is a way to define that the source is required, in which case in case of a missing source table the step will fail and report an error. More information about setting up required steps can be found in the following chapter: Optional and required tables.

By default, a comma (,) is used as a separator. You can define your own separator by overriding the split property. By default, split is set to vertical (meaning that a row will be split into some more rows, this is currently the only value supported). To change the separator, use object notation:

```
{
      "id": 1,
      "name": "Unpivot cars table",
      "type": "split",
      "source": "Cars_Pivot",
      "target": "Cars_Split",
      "column": "cars",
      "split": { "type": "vertical", "separator": ";" }
    }
```

The separator can have more than one character. Bear in mind, that white-spaces are not trimmed, so it is generally up to you.

Example

Given the following input table **Cars_Pivot**:

Id	Name	Cars
1	Marcin	BMW,Renault
2	Juergen	Open,Peugeot
3	Ali	VW
4	Denis	Audi

And the following JSON definition:

```
{
  "id": 1,
  "name": "Unpivot cars table",
  "type": "split",
  "source": "Cars_Pivot",
  "target": "Cars_Split",
  "column": "cars"
}
```

The following table **Cars_Split** is created:

Id	Name	Cars
1	Marcin	BMW
1	Marcin	Renault
2	Juergen	Open

Id	Name	Cars
2	Juergen	Peugeot
3	Ali	VW
4	Denis	Audi

Note:

There is a special chapter in the **Enrichment** section (**Split lookups**) which shows a similar use-case of splitting the value in a column to make a look-up queries. It is recommended to use that approach if the table is used solely for the enrichment purpose.

Enriching

Enriching is an operation which does not translate directly to any SQL query. The basic principle of enrichment relies on look-up, where a selected table (enriched table) is being updated by writing values to one of its columns (enriched column), based on values found in other table.

Source is by default optional, meaning that the table does not have to be existing. If the table is missing, the step will not be executed and its target table will not be written. There is a way to define that the source is required, in which case in case of a missing source table the step will fail and report an error. More information about setting up required steps can be found in the following chapter: Optional and required tables.

Example

For this example, let's consider three tables:

- Table **People** contains the list of people, with an unique ID, name, address and the car plates.
- Table **Car** contains a mapping between car registration and the address of its owner
- Table **Address** contains a mapping between the name and the address.

Table **People**

Id	Name	Address	CarRegistration
1	Marcin	Paderborn	PB-OT-123

Id	Name	Address	CarRegistration
2	John	*<NULL>*	PB-JO-123
3	Andreas	*<NULL>*	*<NULL>*

Table **Car**

CarRegistration	Address
PB-OT-123	Szczecin
PB-JO-123	Berlin
PB-JK-123	Lichtenau

Table **Address**

Name	Address
Piotr	Krakau
Adrian	Stettin
Simon	Dortmund

The problem that needs to be solved is the column **Address** in the table **People**. We want to enrich that table by adding values to the columns that do not have them yet. However, the other tables do not provide an easy matching via JOIN operator, which is where the enrichment kicks in.

The basic idea is:

1. If there is already an address in the table **People** we take it as-is. Otherwise, go to the next point.
2. For remaining rows with still empty value of the address, if there is an entry in the **Address** table, where the value of the column **Name** matches the value in the **People** table / **Name** column, then we should take it from there. Otherwise, go to the next point.
3. For remaining rows with still empty value of the address, if there is an entry in the **Car** table, where the value of the column **CarRegistration** matches the value in the **People** table / **CarRegistration** column, then we should take it from there. Otherwise, go to the next point.
4. If there are still rows with missing address in the **People** table, we take **n/a** as a placeholder.

This operation can be achieved by the following JSON definition:

```
    {
      "id": 1,
      "name": "Enrich People with other tables",
      "type": "enrich",
      "source": "People",
      "target": "People_enriched",
      "column": "Address", // the column to be enriched - point (1)
      "lookup": [
        {
          "in": "Car", // point (2) - lookup [People].[Car]
          "key": "CarRegistration", // matching the value from [Car].[CarRegistration]
column.
          "take": "Address" // if found, take [Car].[Address]
        },
        {
          "in": "Address", // point (3) - lookup [People].[CarRegistration]
          "key": "Name", // matching the value from [Car].[CarRegistration] column.
          "take": "Address" // if found, take [Address].[Address]
        },
        {
          "take": {
            "value": "n/a" // point (4) - if all the previous attempts did not
resulted in a non-null value, then use n/a.
          }
        }
      ]
    }
```

As such, the lookups property is an array of objects which have one of the following structures:

- An object defining the **in**, **key** and **take** properties for look-ups
- An object defining only the **value** table for a fall-back.

The lookup items are ordered and used from the top to the bottom. Once a value for the row can be found from a look-up, it is not going to be looked-up in other sources (for that row). The element in is the look-up source. It can be another step (referenced by ID, see chapter Chaining steps) or a table name (string).

Note:

Enrichment does not override the values in the enriched columns that are already not null. If you want the results of the look-up to be not affected by the state of the current table, use a column name that does not exist in the first table.

Once the above JSON is started, the following table **People_Enriched** is returned:

Id	Name	Address	CarRegistration
1	Marcin	Paderborn	PB-OT-123

Id	Name	Address	CarRegistration
2	John	Berlin	PB-JO-123
3	Andreas	Stettin	*<NULL>*

Enriched values are highlighted using the yellow color to show the differences between the original table.

Look-up values

The look-up target (property take) is using the same syntax as the column mapping. It supports however a limited sets of functions, including taking values as-is, transforming them or using fixed values.

Note:
Enrichment take element does not support Auto expressions! You can use SQL expression though if you need the greatest flexibility.

Due to this, it is possible to use the following combinations:

Taking value from a column without transforming it

```
{
    "in": "Car",
    "key": "CarRegistrationId",
    "take": "Address"
}
```

Taking value from a column with a transform (for example uppercase)

```
{
    "in": {
        "name": "Car",
        "transform": "Uppercase"
    },
    "key": "CarRegistrationId",
    "take": "Address"
}
```

Note:
It makes no sense to use fixed values for non-fallback values. If the value is fixed and is not-null, all subsequent look-up items will always be ignored.

Split look-up

Split look-up describes the following scenario:

- There is an input table to be enriched with data from another table(s).

- There is another table which has the information to be enriched with.
- However, the second table does not have a normalized column that could be used for matching, because its matching key candidates are aggregated - for example separated by an arbitrary separator.

Consider the following table:

Table **EComputer**

Computer	IpAddress	Domain
HOST1	192.168.170.1	DomainA
HOST2	192.168.170.2	*<null>*
HOST3	192.168.170.6	*<null>*

Table **EAddressDomain**

IpAddress	Domain
192.168.170.1; 192.168.170.2	DomainB
192.168.170.3; 192.168.170.4	DomainC
192.168.170.5	DomainD

We want to enrich the first table, so that its *<null>* values in the **Domain** column are set to matching values from **EAddressDomain** (column **Domain**). To identify and lookup a proper device, we use its IP address. However, there is no straightforward way to do this, because the table **EAddressDomain** contains aggregated lookup keys, in form of the column **IpAddress**, where the values are separated with *"; "*(semicolon followed by space).

A workaround would be to use another **Split step** to convert the second table to a normalized version, and then continue with the enrichment. The same can be achieved much quicker and with a less verbose way, by using an automatic cell separator.

```
{
      "id": 1,
      "name": "Enrich EComputer with EAddressDomain",
      "type": "enrich",
      "source": "EComputer",
      "target": "EComputer_enriched",
      "column": "Domain",
      "lookups": [
```

```
        {
          "in": "EAddressDomain",
          "key": "IpAddress",
          "separator": "; ",
          "take": "Domain"
        },
        {
          "take": { "value": "n/a" }
        }
      ]
    }S
```

After adding a separator and executing the JSON, the following table will be created:

Table **EComputer_enriched**

Computer	IpAddress	Domain
HOST1	192.168.170.1	DomainA
HOST2	192.168.170.2	DomainB
HOST3	192.168.170.6	n/a

Remarks and points of interest:

- HOST1 remained unchanged, because it already had a value in the **Domain** column.
- HOST2 received a value of **DomainB**, because its IP address 192.168.170.2 has been found in the list of IP addresses in the second table.
- HOST3 has received a value of **n/a**, because its IP address was not found in any of the list of IP addresses in the second table. More details about fallbacks can be **found in this section**.

Look-up keys

The value of key parameter is the name of the column used for look-up. If you provide it as a string, it is assumed the column has the same name in both main table and the table being used for look-ups. If the columns have different names, use the following syntax:

```
{
    "in": "Car",
    "key": {
        "primary": "CarRegistrationId",
        "foreign": "CarRegistration_FK"
    },
    "take": "Address"
}
```

Where the value for primary property is the name of the column in the main table (enriched table), and the value of the foreign property is the equivalent column in the table used as a look-

up table.

Fallback

The last item in the enrichment list may be a fallback, which provides a way to guarantee there is a non-null value in the enriched column once the transformation is over.

```
{
  "take": {
     "value": "n/a"
  }
}
```

The fallback is defined by omitting any parameters except of the **value**, which should be a fixed value to be used as a last-resort if all enrichment steps failed to deliver a non-null value.

You can safely omit the last step (fallback). If fallback is not defined (all look-up items have parameter in) then NULL will be used as a fallback value.

Value types

The value type is inferred from the context. To enforce an arbitrary column type (for example `VARCHAR(128)`) use the property *type*. More information about arbitrary types can be found in chapter Arbitrary column types

Consolidating look-up targets

If all your tables have the same look-up column and key, you can use an array for the value of in parameter. For example, the following two statements are equal:

```
"lookup": [
    {
        "in": "Cars1",
        "key": "CarRegistration",
        "take": "Address"
    },
    {
        "in": "Cars2",
        "key": "CarRegistration",
        "take": "Address"
    }
]
```

Is equal to the following (consolidated in property):

```
"lookup": [
    {
        "in": ["Cars1", "Cars2"],
        "key": "CarRegistration",
        "take": "Address"
    }
]
```

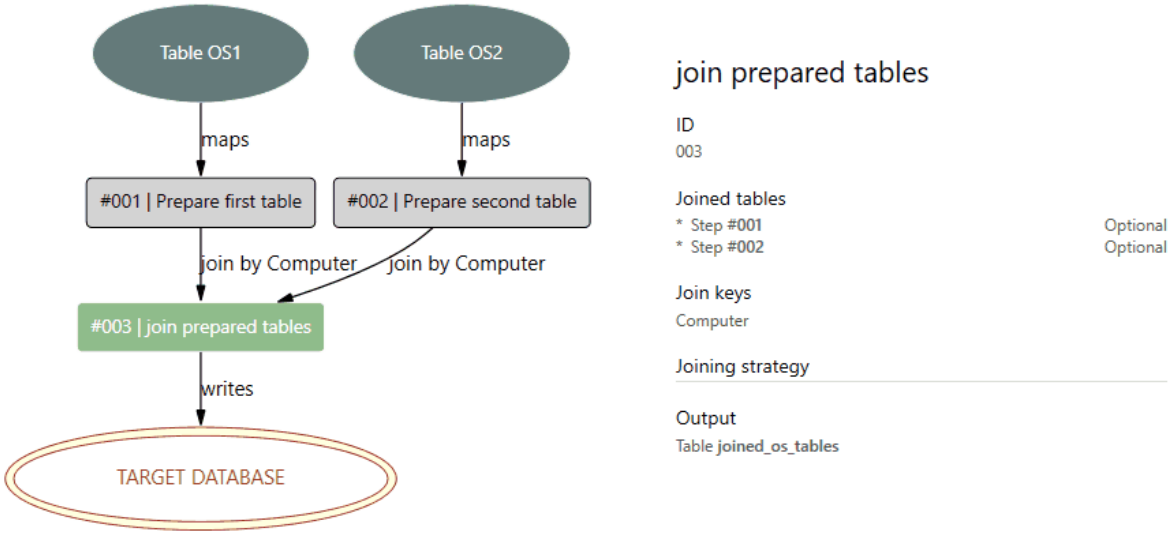
The priority of lookups is then defined by the order in which they appear in the **in** statement.

Chaining steps

Frequently, a step is just an intermediary operation for another steps. For example, having two raw tables, before we can join them at all we need to make sure that:

- Both have normalized columns, including joining keys which must have the same names
- That the values in these columns are normalized and comparable
- That we only have the columns that we need

In this typical scenario, the user would introduce first a single mapping step for each table (to prepare its data), and then the second actual join step which uses the prepared tables. In workflow editor, this is represented by the following schema:



You can define as many steps as required, and just tell ETL engine how to link them (chain) together. The steps are never executed in the order they appear in JSON file, but more like in the actual order determined by the set of dependencies and inter-links.

Previously, we defined sources for steps linking to source tables, for example:

```
"steps": [
    {
        "id": 2,
        "name": "Normalize Sample",
        "type": "map",
        "source": "sample",
        [...]
```

In order to define a source to be another step, simply write its ID instead:

```
"steps": [
    {
        "id": 2,
        "name": "Normalize Sample",
        "type": "map",
        "source": 1,
        [...]
```

Using this example, output of step 1 will be used as an input for step 2.

The same principle follows for the join tables:

```
{
    "id": 3,
    "type": "join",
    "name": "join prepared tables",
    "sources": [1, 2],
    "on": ["Computer"],
    "strategy": "outer",
    "target": "joined_os_tables"
}
```

You can also combine steps and source tables, but since JSON array may contain either strings or numbers, there is a special syntax for that. To combine table abc, def and results of steps 1 and 2 use the following:

```
{
    "id": 3,
    "type": "join",
    "name": "join prepared tables",
    "sources": [
        {
            "table": "abc"
        },
        {
            "table": "def"
        },
        {
            "step": 1
        },
        {
            "step": 2
        }
    ],
    "on": ["Computer"],
    "strategy": "outer",
    "target": "joined_os_tables"
```

```
}
```

Optional and required tables

In many places, it is possible to define a source table or a source step for operations (for example a source table for map step, source tables for join/enrich operations etc.).

Unless otherwise specified, the source is by default optional. This means that the following two examples:

```
{
  "id": 1, // unique ID
  "type": "map",
  "name": "Description of the step",
  "source": "Sourcetable",
  "columns": {
    // a dictionary of columns
    [...]
  },
  "target": "The name of the output table"
}
```

or

```
{
  "id": 1, // unique ID
  "type": "map",
  "name": "Description of the step",
  "source": 1, // step id
  "columns": {
    // a dictionary of columns
    [...]
  },
  "target": "The name of the output table"
}
```

will not fail if the table Sourcetable or table created by step 1 do not exist.

To mark a table or a step as "required", use the object notation in the following form:

```
{
  "id": 1, // unique ID
  "type": "map",
  "name": "Description of the step",
  "source": {
      "step": "Sourcetable",
      "required": true
  },
  "columns": {
    // a dictionary of columns
    [...]
  },
  "target": "The name of the output table"
}
```

or

```
{
  "id": 1, // unique ID
  "type": "map",
  "name": "Description of the step",
  "source": {
      "step": 1, // step id
      "required": true
  },
  "columns": {
    // a dictionary of columns
    [...]
  },
  "target": "The name of the output table"
}
```

By setting the property `required` to `true`, the ETL engine will check if the table exists before starting the execution.

The following rules apply:

- If the source is a required table and the table does not exist in the source, the process will throw an exception before it is even started.
- If the source is a required wildcard, then at least one table matching the pattern must exist in the source, otherwise the process will throw an exception before it is even started.
- If the source is a required step, then the referenced step must produce a table. Some steps will not produce any output if their dependencies are not satisfied. For example, if your step transforms the data from a filter step, then it implicitly means that the source table for that filter step must exist, otherwise the filter step will not produce any value. More information about linking the steps together can be found in the following chapter: Chaining steps.

Programmability

This chapter discusses how to extend the ETL engine with SQL snippets.

SQL environment

Creating reusable scripts

To promote usability and enable easier testing of SQL expressions, it is possible to encapsulate SQL snippets and store them separately from the JSON definition.

Once the ETL engine loads the ETL definition, it looks for the file which has the same file name as your JSON definition but with extension .SQL. If it finds it, it uses its content in all sessions created for evaluation of SQL expressions. Writing scripts in that file is much easier than inline expressions in the JSON file, because of support for new lines, lack of necessity to escape special characters.

The format of the .SQL file is not a standard SQL though. Its content should be of the following

syntax, where each function is defined like:

```
def <MacroName>(@<param1>, @<param2>, ...)
    <macro-body>
```

For example, the following is a custom file which contains two macros

```
def capitalize(@val):
    UPPER(substr(@val, 1, 1)) + LOWER(substr(@val, 2))

def isNullOrEmpty(@text):
    @text IS NULL
```

Parameter names must always start with '@' sign. The body can span over several lines, and it must be a valid SQLite syntax. You can reference the parameters, also using the '@' syntax. In this example, macro capitalize takes a value (@val) and returns a capitalized version of it. The other one isNullOrEmpty returns 1 or 0, depending on whether the parameter is null.

In order to use the macro in your JSON code, prepend its name with '!' and followed by the list of arguments, for example:

```
{
    'sql': !capitalize(name)'
}
```

Macros support column names and expressions. Bear in mind, that commas (,) have a special meaning, they separate the arguments. This is why the following will fail:

```
{
    'sql': !capitalize(SELECT col1, col2 FROM table)'
}
```

Instead, enclose the expression with braces to indicate that the commas belong to the argument.

Recursive calls

Macros can be called recursively (a macro calls another macro or even itself). Bear in mind that ETL engine limits the number of recursion levels.

Extra functions

Note:
These functions are available in ETL module version 1.1.271 and newer.

ETL extends capabilities of SQLite engine by including the following, non-standard functions:

- **SHA1**
  Returns a 40-character string representing 160-bit (20 bytes) SHA1 hash of an input element.

This function requires a single parameter.

- **MD5**
  Returns a 32-character string representing 128-bit (16 bytes) MD5 hash of an input element. This function requires a single parameter.

- **REGEXP**
  Returns true if the value matches a given expression. This functions requires two parameters.

Sample usages:

- `[Mail] REGEXP "@raynet\.(de|ch|com)$"`
  Returns 1 if the value from the *Mail*column matches the given pattern.

- `REGEXP ("@raynet\.(de|ch|com)$", "m.otorowski@raynet.de"`
  (Alternative syntax) Returns 1 if the value from the *Mail*column matches the given pattern.

- `SHA1([Mail)`
  Returns 40-byte SHA-1 hash of the value from the *Mail*column.

Special characters and escaping sequences in Regular Expression patterns

Regular expressions use \ (backslash) as an escape sequence. Bear in mind the same escape sequence is used by JSON schema. This means, that when escaping a regular expression inside a JSON file, it is necessary to perform "double-escaping".

For example, to test whether the name consists of letters, followed by a dot, followed by a number, the following Regular Expression can be used:

```
^[a-zA-Z]+\.[0-9]+$
```

Explanation:

- `^` means this is the beginning of the string
- `$` means the string ends here
- `[a-zA-Z]+` - one or more instances of characters from range a-z or A-Z.
- `[0-9]+` - one or more digit
- `\.` - literal dot. Dot has a special meaning in Regular Expressions (any character), and to use it literally it must be escaped with `\` (backslash).

However, the backslash in the regular expression must be escaped once more when used in JSON file, so that it is not interpreted as escape JSON sequence. The following JSON would be valid:

```
"sql": "[columnName] REGEXP '^[a-zA-Z]+\\.[0-9]+$'"
```

Additional Information

Visit *www.raynet.de* for further information on ETL, and take a look at the additional resources available at the Knowledge Base: http://raynetgmbh.zendesk.com/.

Raynet is looking forward to receiving your feedback from your ETL experience. Please contact your Raynet service partner or use our Support Panel to add your ideas or requirements to the ETL development roadmap!

ETL
is part of the
RaySuite

More information online
www.raynet.de

Raynet GmbH
Technologiepark 22
33100 Paderborn, Germany
T +49 5251 54009-0
F +49 5251 54009-29
info@raynet.de

www.raynet.de