HITACHI
Inspire the Next

Pentaho Data Integration (PDI) Techniques - Standards for Lookups, Joins, and Subroutines

HITACHI
Inspire the Next

Change log (if you want to use it):

Date	Version	Author	Changes
10/11/2017	1.0	Matthew Casper	

Contents

This page intentionally left blank.

Overview

This document covers some best practices on Pentaho Data Integration (PDI) lookups, joins, and subroutines.

Our intended audience is PDI users or anyone with a background in ETL development who is interested in learning PDI development patterns.

The intention of this document is to speak about topics generally; however, these are the specific versions covered here:

Software	Version(s)
Pentaho	7.x, 8.0

The Components Reference in Pentaho Documentation has a complete list of supported software and hardware.

Before You Begin

Before beginning, use the following information to prepare for the procedures described in the main section of the document.

Other Prerequisites

This document assumes that you have knowledge about PDI and that you have already installed Pentaho software.

Use Cases

Use cases employed in this document include the following:

- *An end user of Pentaho Data Integration wants to understand when it is appropriate to use a Stream Lookup versus a Database Lookup for dimensional data prior to loading a database fact table.*
- *A telecommunications company is looking to replace redundant code lines with consistent processes but needs to know if the solution is better suited for a Mapping subroutine or a Transformation Executor.*

Lookups

In this section, we will explore four of the more common ways to look up data from within a transformation and the standards and best practices associated with each. In all four cases, there are ideal situations for arguing their use over the other options:

You can find details on these topics in the following sections:

Database Join

The **database join** step offers some unique abilities to look up data in one or more database tables. This step allows for the abilities to:

- Write custom SQL
- Return the fields desired, which can be calculations using functions
- Pass in variables
- Use parameters as arguments in the SQL
- Join across multiple tables
- Write subqueries
- Perform outer joins

Despite these advantages, there are two main performance issues to consider:

1. You cannot cache data in a **database join** step as in other lookup step types. Caching data allows you to store records in memory instead of fetching against the database, which provides excellent performance.
1. The custom SQL is performed for every row that is passed into the step, so if 1,000 rows are passed into the step, that means there will be 1,000 queries performed against the database. Because of this, you need fast-running, simple SQL that has been performance tuned (such as with column indexing) to effectively use a **database join** step.

Figure 5 shows a database join step example. In it, we needed to apply logic to three incoming date fields used as parameters (?s in the SQL) to get the correct foreign key (`dim_date_hour_key`) and date (`date_value`).

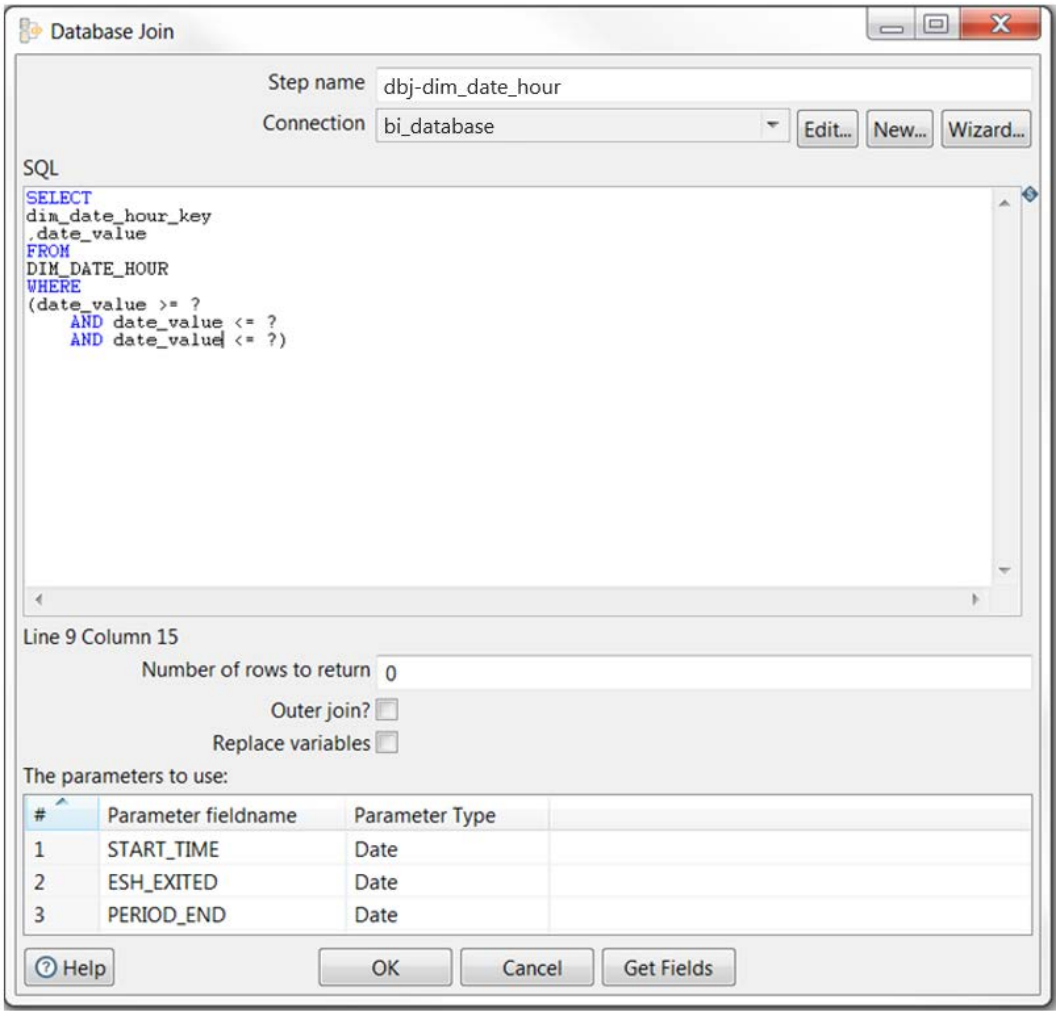

Figure 5: Database Join Step Example

Aside from the performance issues discussed earlier, you must also know when it is appropriate to use this step, or other steps. Your logical lookup rules will help you make this decision. Do you need to:

- Join more than one table together?
- Apply a database function to the returning field?
- Apply more intricate `WHERE` clauses involving `OR` conditions or subqueries?

Whatever your rules may be, testing will help you see which lookup steps work best for your situation. Try doing a comparison with realistic datasets before you decide on using a **database join** step.

Table 5: Should I Use a Database Join Step?

Need	Reasons to Use Database Join
Special lookup logic required	Joins across tables, intricate `WHERE` clause, subqueries
Number of streaming records is relatively small	Performance can be negatively impacted because the step fires the SQL for each streaming record
Need to use variables or parameters	Pass streaming fields in as arguments in the SQL; use variables in the SQL
Special logic for return fields	Database functions need to be applied, aggregates such as `SUM`, `COUNT`, `MAX`, `MIN`

Database Lookup

The **database lookup** step functions in much the same way as the database join, because you can use different comparison operators like greater than/less than/equal to, `BETWEEN`, `LIKE`, `IS NULL`, and `IS NOT NULL`.

Two things that cannot be done within the database lookup are:

- Custom SQL
- Applying database functions

Despite these limitations, a big performance boost is available to the **database lookup** step in the form of the **Enable cache** option. This option allows you to store each record that meets the join criteria in memory instead of constantly referring back to the database as in the **database join** step.

If you enable the cache, another option to **Load all data from table** becomes available. This can also help you boost your performance because the step will collect all records from the lookup table and store them in memory, holding them available for immediate access.

To use this option, first make sure that all the records of the table will fit in memory, or you could run into Java heap space errors.

In the following database lookup example, we are grabbing the `rental_hourly_rate` from the `dim_rental` table, giving the field a new name (`RENTAL_EXPENSE`), assigning the value of `0` if a match is not found, and setting the **Type** to `BigNumber`. Because this is a small cache and can fit into our

allotted memory slot of 6GB, we will use the **Enable cache** and **Load all data from table** options:

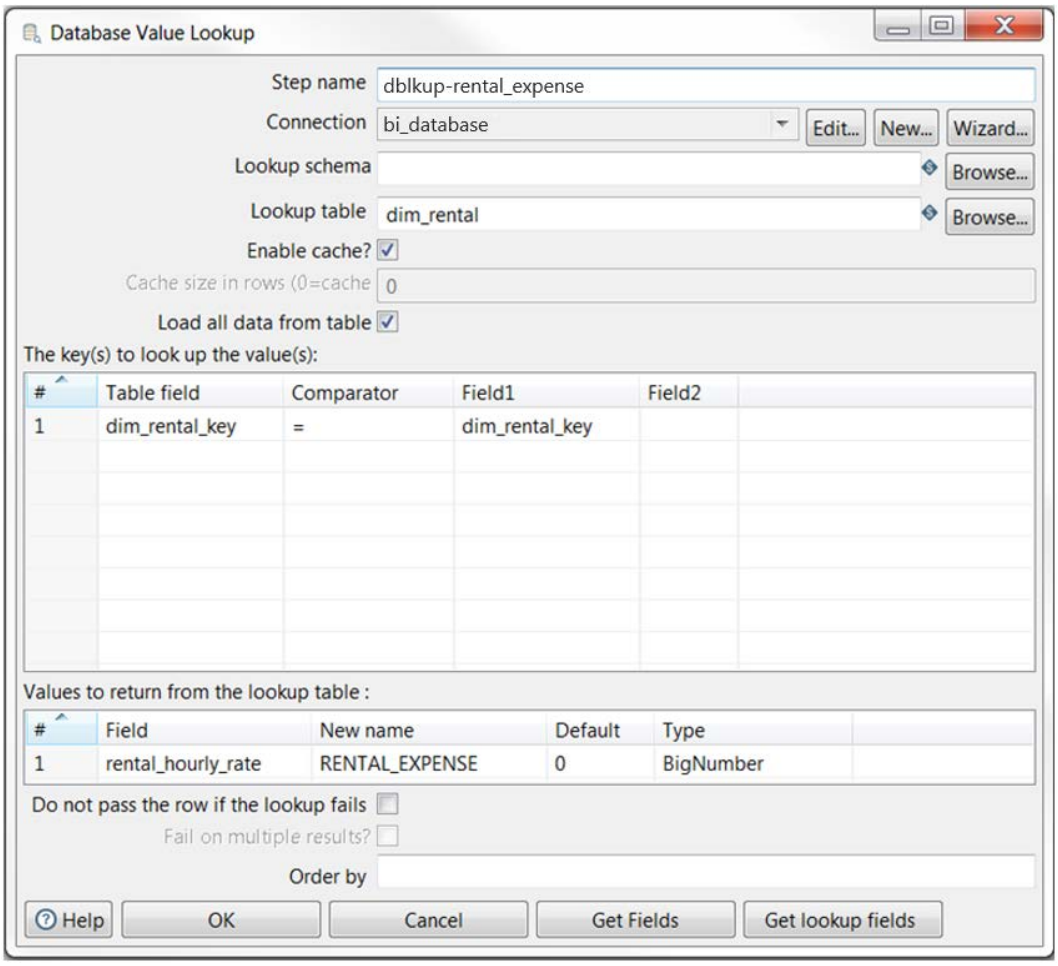

Figure 6: Database Lookup Example

If you have enough memory:

- **Enable cache** will always boost your performance.
- **Enable cache** and **Load all data from table** will boost your performance even more.

We recommend you test with real-world datasets to find your best performance path.

Table 6: Should I Use a Database Lookup Step?

Reasons to Use Database Lookup	Details
Straightforward lookup using base comparisons	Fields using =, >, >=, <, <=, <>, `LIKE`, `BETWEEN`, `IS NULL`, `IS NOT NULL`
Ability to enable the cache	Enabling the cache stores matched records in memory for quicker access to future streaming records.
All data from lookup can fit in memory	The **Load all data from table** option will put all lookup records in memory, providing for substantial performance improvement over going to the database for each record.
Outer join is needed	By default, this is an outer join, allowing the developer to work with the records that don't have a match. However, there is the option to make it an equijoin by enabling **Do not pass the row if the lookup fails**.

Stream Lookup

The **Stream lookup** step is like a combination of the Database join and Database lookup steps. Stream lookup expects an input from the main stream and one from a new input like a **Table Input**.

- Like with the **Database join** step, the **Table Input** can have custom SQL and employ all types of database options and functions.
- Like the **Database lookup** step, Stream lookup will store all the records from that **Table Input** in memory for access, so it will only need to go to the database once.

Things to keep in mind when you consider a **Stream lookup** step include:

- The step always performs an outer join to the **Lookup** step (such as **Table Input**), so you should always configure a default value for the field(s) to retrieve.
- There is no option for you to use any comparators. Stream lookup always uses equals (=) between the `field` and `lookupField` properties. This means that those properties and the fact records are stored only in memory will determine whether this step is the right lookup step for your transformation.
- **Preserve memory (costs CPU)** is checked by default, and should be left checked in most cases because the rows of data will be encoded, preserving memory while sorting. However, this effect comes at the expense of the CPU calculating the hashcode.

In the following example, we have our main stream going into the `slkup-shrunk_items` Stream lookup step, and the `ti-shrunk_items` is a **Table Input** defined as the lookup step. In the `ti-shrunk_items`, you have your custom SQL with joins across tables, and `WHERE` clauses. In `slkup-shrunk_items`, notice the section **The key(s) to look up the value(s)** does not provide any comparator options. Finally, in the **Specify the fields to retrieve** section, we provide a new name to

the retrieval field and set a default value. **Preserve memory (costs CPU)** is left enabled:

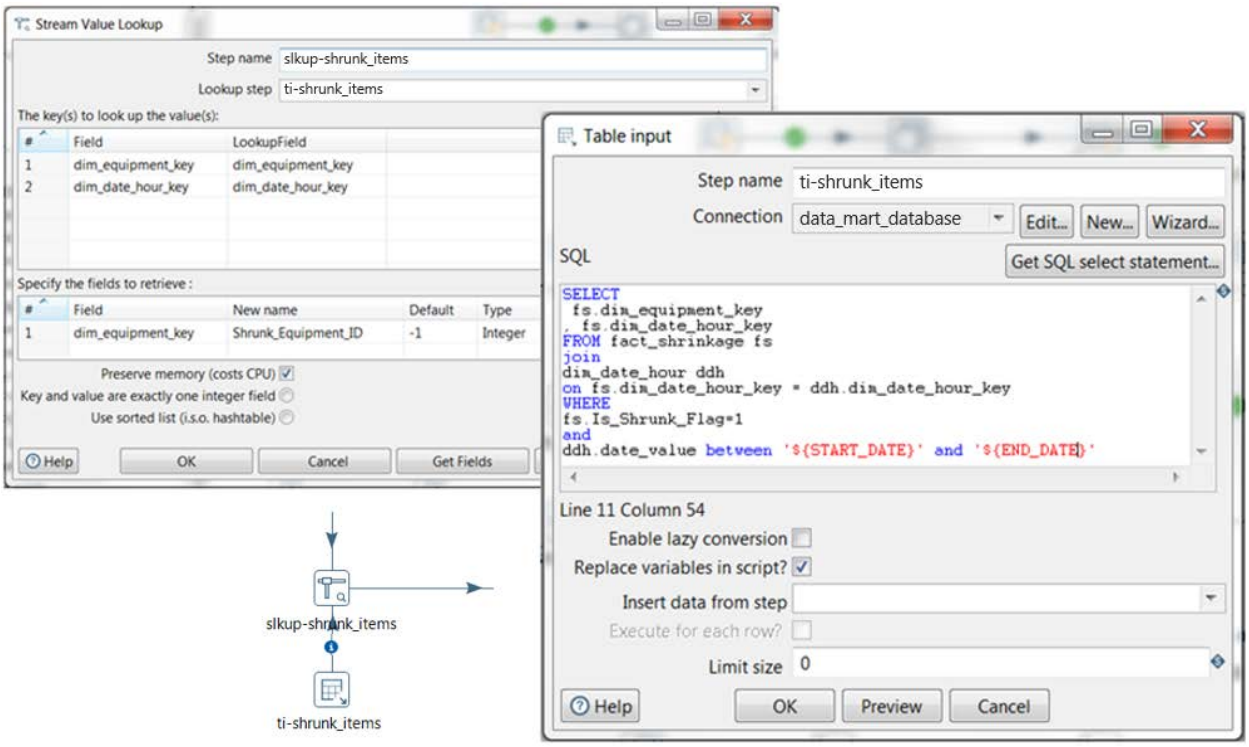

Figure 7: Stream Lookup Example

Table 7: Should I Use a Stream Lookup Step?

Reasons to Use Stream Lookup	Details
No need to use comparators other than =	The step only performs lookups using = and cannot use any other operators.
Ability to preserve memory	Enabling the Preserve memory setting saves memory at the expense of CPU.
All data from the lookup can fit in memory	All data from the **Lookup** step will go into memory, providing for substantial performance benefits over going to the database each time.
Outer join is needed	By default, this is an outer join, allowing the developer to work with the records that don't have a match. We recommend setting values for the `Default` property of fields being retrieved.

Dimension Lookup/Update

The **Dimension lookup/update** step is geared toward dimensions with slowly changing structure, such as `date _from` and `date_to` columns that identify historic or active records.

Generally, the transformation that uses this lookup wants to obtain the dimension table's technical key to be used as a foreign key in the fact table. However, this type of lookup can also be used to obtain other pieces of information from within the dimension table.

Follow these rules when you use this step as a lookup:

1. Uncheck **Update the dimension?** If you leave it checked, the step will be used as a lookup, but will also potentially update records within the dimension.
2. Check **Enable the cache?** This will allow you to store matched or found records in memory, and will increase performance for later streaming records, because the step will only have to go to memory to find records instead of back to the database.
3. Check **Pre-load the cache?** Enable this setting if you are certain that all of the table's records will fit into memory. This will provide you the best and quickest possible performance.

Once you have followed these rules, all that remains is configuring the step correctly for the **Key fields**, **Technical key field**, and **Stream Datefield**. Here is the configuration of all these sections:

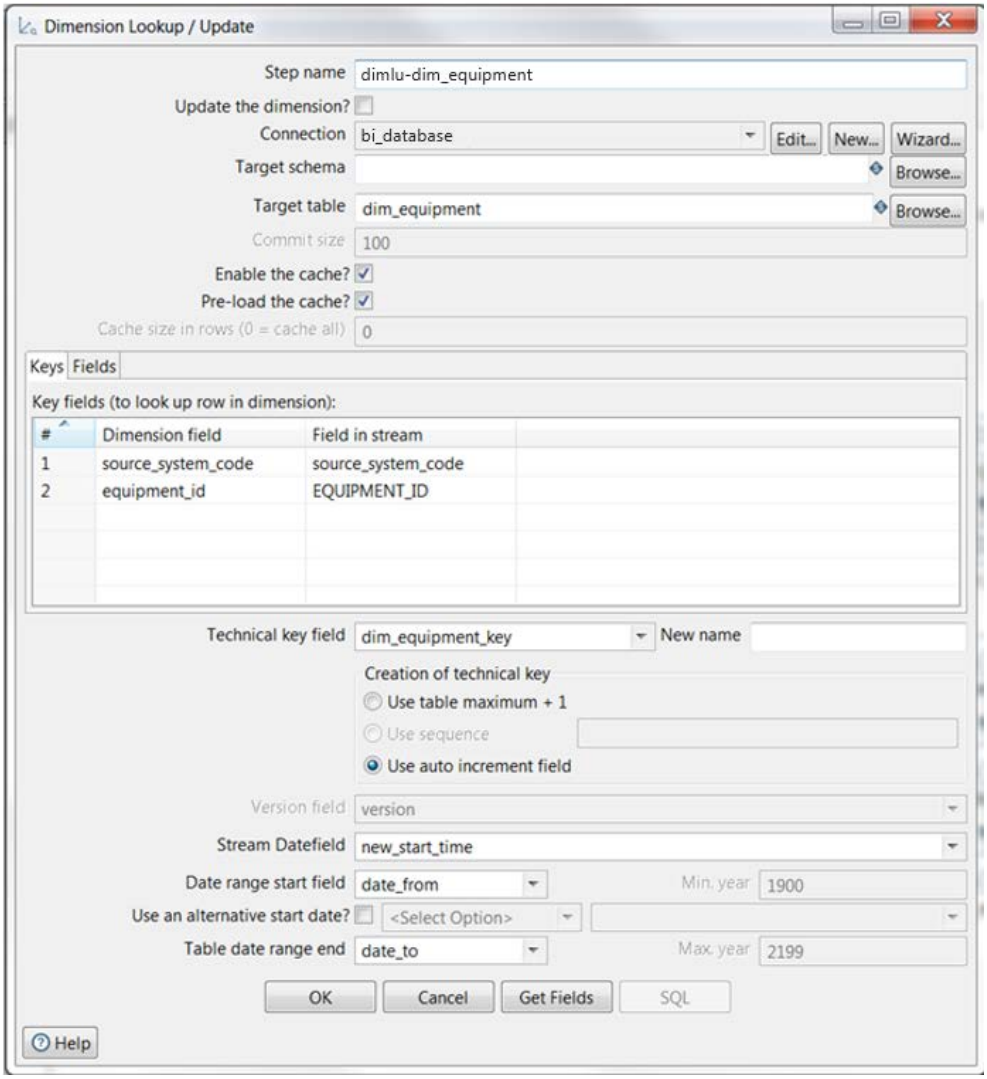

Figure 8: Dimension Lookup/Update Example

Table 8: Should I Use a Dimension Lookup/Update Step?

Reasons to Use Dimension Lookup/Update	Details
Dimension is slowly changing	This step is geared toward dimensions that have slowly changing structure, so if your lookup table does not have that structure, do not use this step.
All data from lookup can fit in memory	All data from the **Lookup** step will go into memory, provided the **Pre-load the cache?** setting is enabled.
No need to use comparators other than =	This step only performs lookups using = and cannot use any other operators.
Outer join is needed	By default, this is an outer join, allowing the developer to work with the records that don't have a match. Unmatched records will receive the stub record values from the dimension table.

Joins

In this section, we will explore joins, which involve bringing together two streams based upon some key logic, or, as in the case of Join Rows (cartesian product), potentially no logic. As with the Lookups in the previous section, a case can be made for using each of these types of join. Your choice will depend on your situation and which join offers you the best performance.

You can find details on these topics in the following sections:

Join Rows (cartesian product)

The first thing to understand about the **Join Rows** (cartesian product) step is that you must use caution when you use this step type. It will certainly join streams together, and you can completely omit the join logic if you want. However, if you have 10 rows from one stream, and 10 rows from another stream, the results of this step will be 100 rows, because it will join for *every* case/record possible.

A potential use for this step is to do data load verification by checking that the number of rows loaded to a table matches the number of rows received in the source file. If these numbers do not match in our comparison, we can take action by sending notifications and/or aborting further downstream processes.

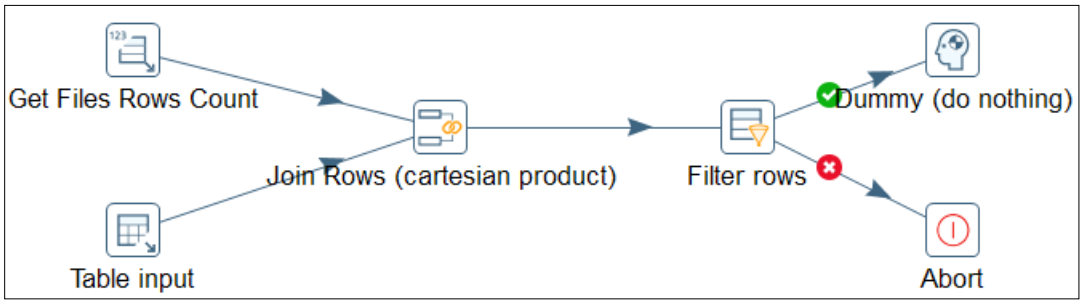

Figure 9: Join Rows (cartesian product) as a Step in a Transformation

Inside the step itself, there is little (or nothing) to do. You do not have to change the Temp directory, the **Main step to read from**, or even set **The condition**:

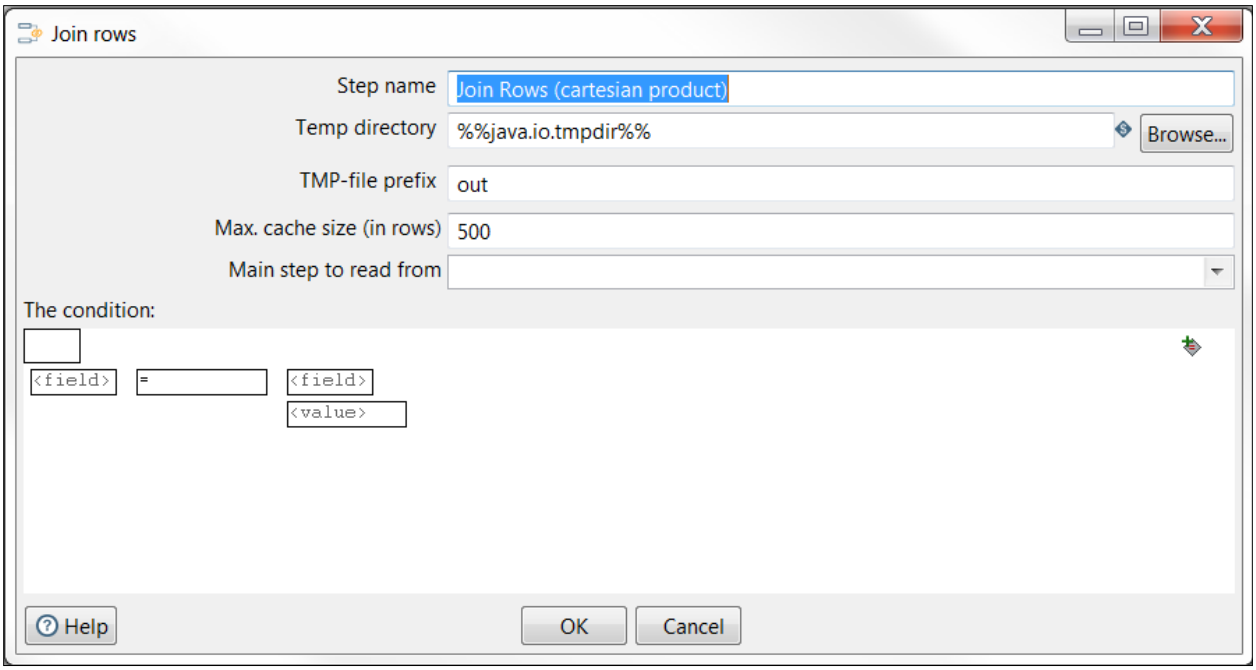

Figure 10: Join Rows Setup

Table 9: Should I Use a Join Rows (cartesian product) Step?

Reasons to Use Join Rows (cartesian product)	Details
Comparing data of disparate sources	If you need to do a record-by-record comparison between data that exists in different sources, this step may be beneficial. See the previous example of comparing records counts between a file and a database table.
Adding constants	This step can be used to add a constant to every record. The constant may come from a different data source in this situation.
Watch record count inflation	This step will apply a join to every record value. If there are 10 records in one stream and 10 in another, you can end up with 100 records, depending on record values.
Applying conditions	Apply conditions such as one date between two others. For instance, a date between start date and end date: `transaction_date` >= `start_date` and `transaction_date <= end_date`.

Merge Join

Merge Join is one of the most common steps to use out of the Joins folder when you want to merge two streams. It simply uses key fields to bring records together, and the join options are `INNER`, `LEFT OUTER`, `RIGHT OUTER`, and `FULL OUTER`.

*You must sort key fields before using this join, so use a Sort Rows or an `ORDER BY` clause in the SQL of a **Table Input** step, or similar.*

One limitation that you may need to address with this step is that in cases where you use a `LEFT` or `RIGHT OUTER` join, you do not have the option of setting a default value, so if there is no match, it returns `NULL`. You may need to account for that if there are `NOT NULL` constraints to your database columns.

Figure 11 shows an example of the **Merge Join** step. Here, we have a main stream that does a Sort Rows prior to the Merge Join, and a **Table Input** representing the second stream, using an `ORDER BY` clause to avoid the need for a Sort Rows in that stream. This example uses `INNER` as its **Join Type**, so if a match is not found, that record is excluded from the stream:

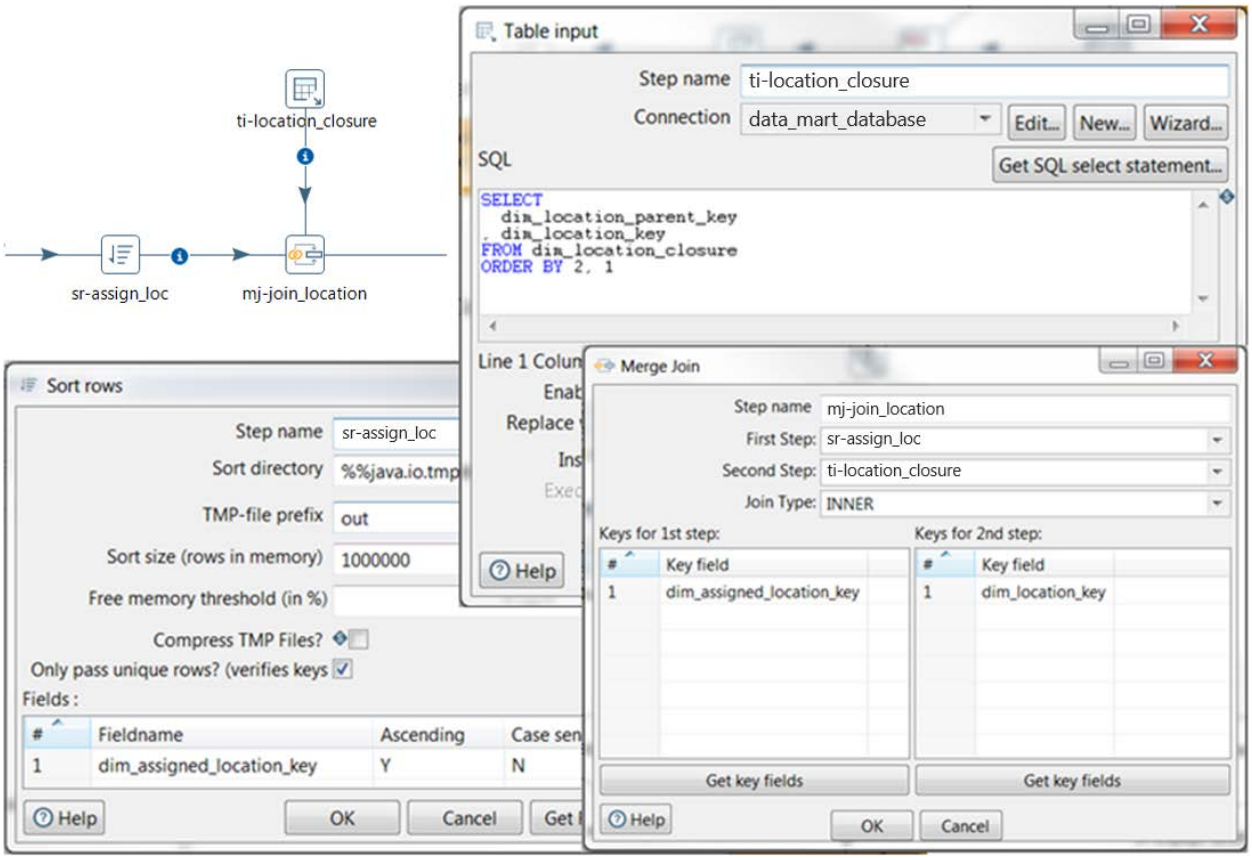

Figure 11: Merge Join Example

Table 10: Should I Use a Merge Join Step?

Reasons to Use Merge Join	Details
Joining data sets of different input steps	This works almost the same as a Stream Lookup, except that records are not stored in memory.
Key fields must be sorted	The step will not work accurately if the key fields are not sorted.
Test for performance	This step can be labor-intensive, so do some testing to be sure it is the right step for your transformation.
Use when memory is a factor	If the environment is such that memory usage is at a premium, using this join step over Stream Lookup can be advantageous.

Merge Rows (diff)

The Merge Rows (diff) step compares two streams of data, creating a "flagfield" that identifies changed data. The flag will have values of `identical`, `deleted`, `changed`, or `new`, and based upon this flag, you can take different actions. For example, you can use a **Switch/Case** step afterward, using the flagfield to route records to be inserted to a table (**Table Output**), updated (Update), deleted (Delete), or do nothing (Dummy).

Much like the **Dimension Lookup/Update** step, Merge Rows (diff) serves as another way to handle slowly changing dimensions. However, this step can be more useful than Dimension Lookup/Update in the case of data sources not having flags, such as update dates, that help identify records that are new or changed. Merge Rows (diff) can compare the source to the target to determine the appropriate actions to take.

In Figure 12, we have two **Table Input** steps, one for the source staging table, and one for the target table. Merge Rows follows, and we have identified the **Reference rows origin**, **Compare rows origin**, **Flag fieldname**, **Keys to match**, and **Values to compare** in the step configuration. Next, we use the **Switch/Case** (`swcs-flagfield`) step to route records based on the flagfield values determined by Merge Rows. Finally, we send the records to the appropriate output step:

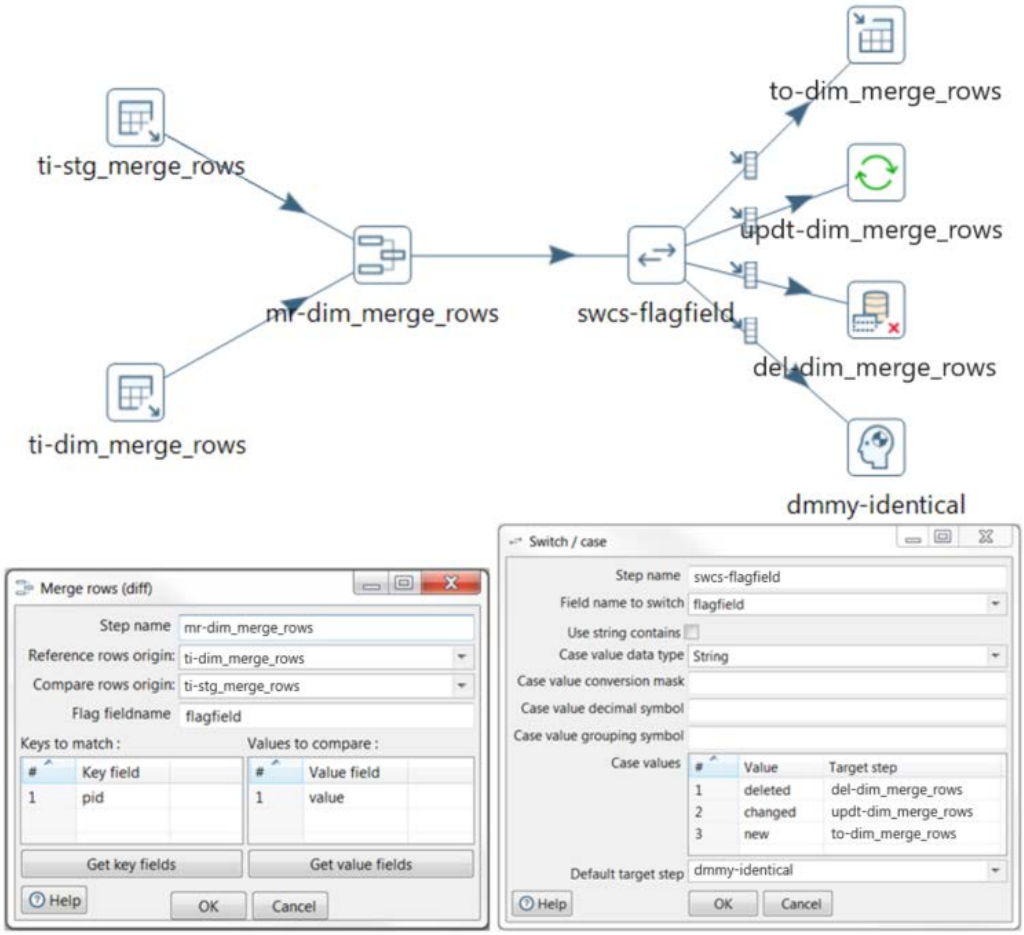

Figure 12: Merge Rows (diff) Example

Table 11: Should I Use a Merge Rows (diff) Step?

Reasons to Use Merge Rows (diff)	Details
Data source does not have an incremental pull indicator	This step is ideal for comparing the source to target data for changed data capture. It is especially useful when the source data structures do not indicate what data is new or changed, preventing you from pulling records incrementally.
Sort required	The Key to match in this step must be sorted, which can affect overall transformation performance especially if there is a large number of records.
Dimension Lookup/Update	Use Dimension Lookup/Update as your first choice for this functionality for a dimension table, but if you need to identify something that has been deleted from the source, Merge Rows (diff) is the best choice.

Subroutines

Subroutines are a good way to minimize the amount of coding that can build up across a project. Generally, subroutines trim down repeated logic into a single routine and minimize otherwise high maintenance costs. If you identify lines of code that can be generalized and functional across many transformations, that may be a good candidate for a subroutine.

Our best practices document on Metadata Injection has specific recommendations for standard, full, and complex Metadata Injection for ETL.

You can find details on these topics in the following sections:

Mappings

If you have a series of transformations that perform the same repetitive steps, you may find mappings useful. With a mapping, you can manage those series of steps in just one place, but apply them across many transformations, reducing your amount of work.

In Figure 13's example, we are receiving a ZIP file of 190 CSV files which each have the filename format of `table_name_start_datetime_end_datetime`. We need to parse each filename; extract the table name, start date/time, and end date/time; and set those extracted values as variables back in the main transformation, to assist in loading our staging database tables. Because every file has the same naming convention, and we need to do this set of actions for every file, we can use a mapping for this series of steps.

The main transformation for the file `equipment_location_history` is in the top left portion of Figure 13. We start with a **Get filenames** step and then pass that to our **Mapping** (sub-transformation) step; the dialog box for this is displayed in the bottom left of Figure 13. There, we have configured the list of fields for the **Input** tab, and just to the right of that is the configuration for the **Output** list of fields we expect to receive back in the main transformation.

The actual mapping itself, beginning with the **Mapping** input specification (`mis-file_metadata`) and ending with the **Mapping** output specification (`mos-stage_transform`), appears in the top right of Figure 13. The last part of the picture is the dialog box for the **Mapping** input specification with the complete list of fields expected to be received along with data types and lengths.

If we were doing this separately for 190 files, if the file naming convention changed, we would have to change 190 separate transformations. However, because we are using a mapping, if the file naming convention changes, there is only one place we will need to adjust, and then that will apply to all 190 transformations:

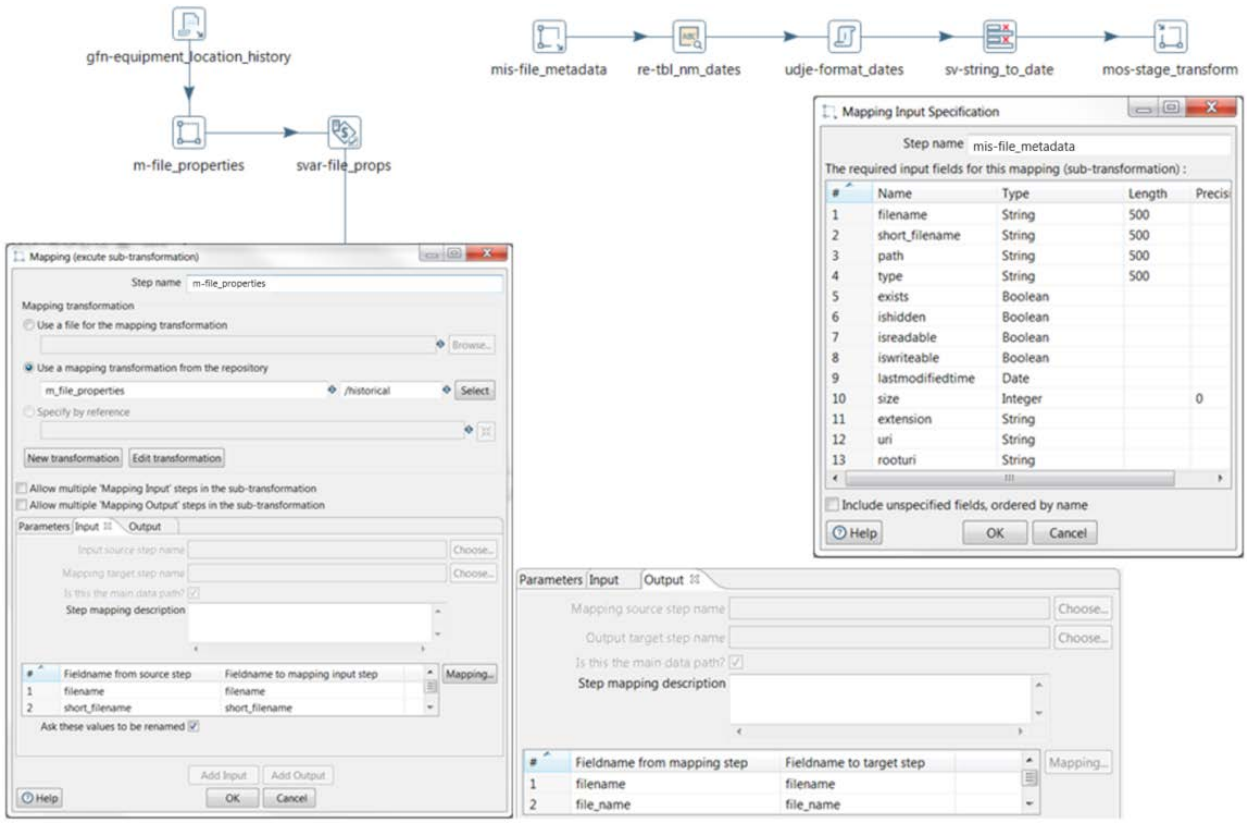

Figure 13: Mapping Example

This is a basic example of using a mapping, because there are only three steps that repeat across all the transformations. There is room for even more logic. In fact, the Mapping (sub-transformation) can be configured to accept multiple inputs and outputs. Our mapping in Figure 13 could be altered to have more than one **Mapping** input specification, and more than one **Mapping** output specification, if necessary.

Table 12: Should I Use a Mapping (sub-transformation) Step?

Reasons to Use Mapping	Details
Series of steps repeated across many transformations	If there is a series of steps repeated across multiple transformations, then a Mapping (sub-transformation) can help lessen maintenance costs.
Parameters can be passed	This step can accept parameters from the main transformation and pass them down to the mapping, if you configure the Parameters tab appropriately.
Multiple Inputs/Outputs can be configured	You are not limited to a single path for the mapping because you can send different inputs to different **Mapping** input specifications, and different outputs to different **Mapping** output specifications.

Transformation Executor

Transformation Executor, like a **Copy Rows to Result** step, performs the subsequent transformation for every input row. However, Transformation Executor lets you do this within your main transformation instead of coming back out to the job level and configuring a following transformation entry.

Features of Transformation Executor include the ability to:

- Send more than one stream to it.
- Launch multiple copies to gain parallel processing.
- Pass step parameters (like for Mapping (sub-transformation)).

The example in Figure 14 is taken from a project where we were serializing up to five different files which all needed to be combined into one file at the end. In the first part, we are doing checks for whether the file exists, using the **File Exists** and **Filter Rows** steps (`fe-bld_mbr_1`, `fr-exists`). Then, because the top two streams have the same desired file input and output structure, they each lead into their own Transformation Executor (`te-deserialize_member`).

The bottom left of Figure 14 displays the dialog box for this step. We are passing a parameter, `FILE_NAME_SER`, and in the middle right of Figure 14 we have configured our **Result rows** tab, stipulating that **The target step for output rows** is `ac-loop_order_placeholder`. The expected list of fields from the result rows is listed below that.

Figure 14's bottom right shows the **Result rows** tab configured for `te-deserialize_member_ben`. Notice that its target step is `sv-mbr_ben_to_sort`. That is all we need to configure for these Transformation Executors. We did increase the number of copies to start for each to be x2 and x3, to obtain that parallel processing advantage.

Finally, the transformation that the Transformation Executor points to is very simple. See the top right of Figure 14. We are taking the parameter being passed, `${FILE_NAME_SER}`, into the **Deserialize from file** step, and then copying the rows back out to the main transformation for processing.

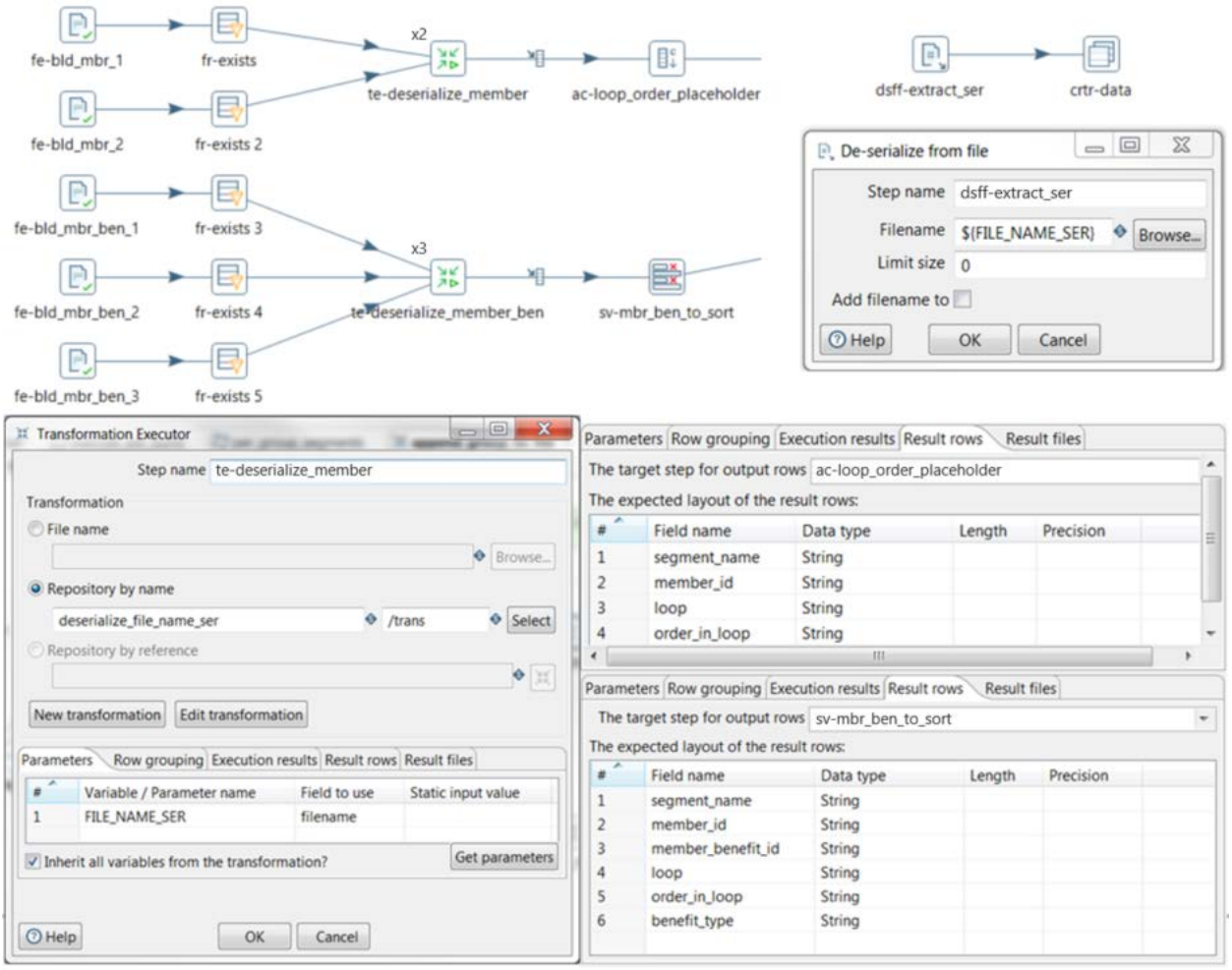

Figure 14: Transformation Executor Example

In certain cases, you may not need to come back out to the main transformation for further processing. The code line can certainly end within the transformation that the Transformation Executor is referencing. You would still need to configure the **Result rows** tab in the main transformation, but you can set that up to point to a **Dummy** (do nothing) step:

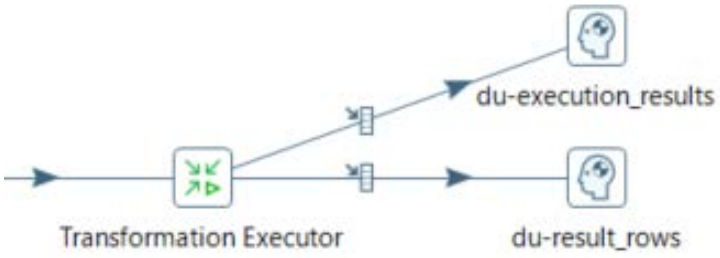

Figure 15: Transformation Executor Example 2

Table 13: Should I Use a Transformation Executor Step?

Reasons to Use Transformation Executor	Details
Replaces Copy Rows to Result	Consider using Transformation Executor in cases where you would otherwise use a **Copy Rows to Result** step.
Parameters can be passed	This step can accept parameters from the main transformation, and pass them down to the called transformation. Configure the **Parameters** tab to do this.
Configure Execution results for error handling	If the called transformation has an error, this step will not abort. The tab of Execution results must be configured for error handling. This includes having access to the full log for the called transformation.
Handles more than one stream	If the record structures are the same, this step can ingest more than one stream at a time.
Parallel processing	The step allows for launching more than one copy to start, to take advantage of parallel processing.

Job Executor

Job Executor, like a **Copy Rows to Result** step, performs the subsequent job for every input row. However, Job Executor lets you do this within your main transformation instead of coming back out to the job level and configuring an ensuing job entry.

Job Executor works just like, and has all the same options as, Transformation Executor. The only difference is that the step launches a job rather than a transformation.

In the example in Figure 16, we are tasked to perform a migration process that moves data from a set of MySQL tables and loads it to the same tables in MSSQL, with only slight differences. The top portion of the figure represents the main transformation where we use a **Table Input** to pull metadata from the MSSQL `information_schema_columns` table. This information drives the construction of the SQL statement used to pull from MySQL.

Next, we group the column names separated by a comma by the table name, and send the columns and table name into the `j_mysql_admin_migration` Job Executor as parameters. All of the processing for pulling data from MySQL and loading it into MSSQL will occur in the called job `j_mysql_admin_migration`; therefore, we do not need to configure the **Execution Results** and **Result Rows** tabs.

Lastly, in the called job, we write to the log `wtl-table_load` to document what table is being processed, and use a transformation `t_mysql_admin_migrate_tbls` to receive the columns and table parameters to pull the data from MySQL and write to a text file. Then, we bulk load the text file to MSSQL with `sql-bulk_load_mssql`, and finally remove the text file that was created for that table with `df-table_name_csv`.

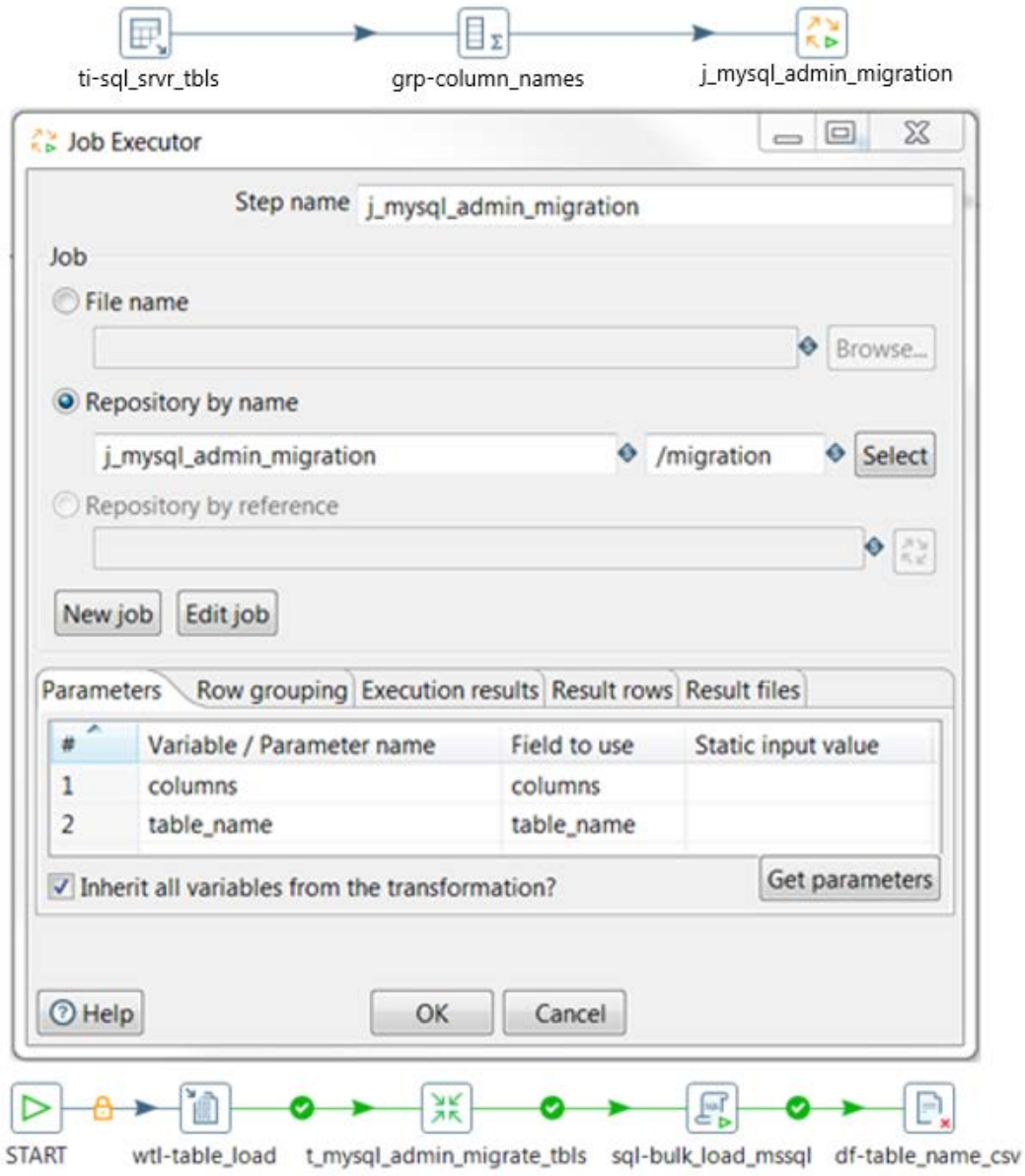

Figure 16: Job Executor Example

Table 14: Should I Use a Job Executor Step?

Reasons to Use Job Executor	Details
Can replace Copy Rows to Result	Consider using Job Executor in cases where you would otherwise use a **Copy Rows to Result** step.
Parameters can be passed	This step can accept parameters from the main transformation and pass them down to the called transformation. Configure the **Parameters** tab for this. The checkbox **Inherit all variables from the transformation** means that you do not have to list the variables here; they will be passed down.
Configure Execution results for error handling	If the called job has an error, this step will not abort. The Execution results tab must be configured for error handling. This includes having access to the full log for the called job.
Handles more than one stream	If the record structures are the same, this step can ingest more than one stream at a time.
Parallel processing	This step allows for launching more than one copy to start to take advantage of parallel processing.

Related Information

Here are some links to information that you may find helpful while using this best practices document:

- Components Reference
- Best Practices for Metadata Injection

Finalization Checklist

This checklist is designed to be added to any implemented project that uses this collection of best practices, to verify that all items have been considered and reviews have been performed. (Compose specific questions about the topics in the document and put them in the table.)

Name of the Project:_____

Date of the Review:_____

Name of the Reviewer:_____

Item	Response	Comments
Did you choose the best lookup method for your purposes?	YES_____ NO_____	
Did you choose the best join method for your purposes?	YES_____ NO_____	
Did you consider subroutines, and use them if possible?	YES_____ NO_____	