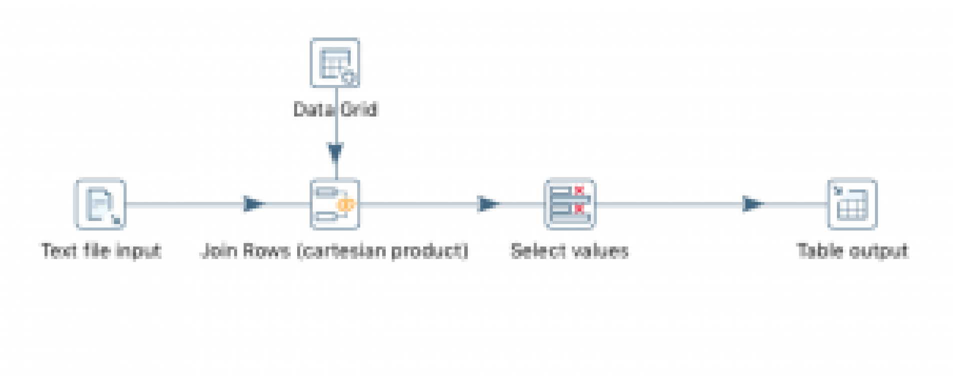
Loading CSV files with PDI Metadata Injection



Introduction

We often need to develop PDI jobs and transformations to load CSV files. In an ideal world, any data source will provide us with a single format for ingestion. All files look the same and a single PDI transformation can load them all into a database table.

But as is often the case, some files will have slight differences. Or not so slight ones. We've faced on several occasions the need to ingest files where the number of fields varies; or the order in which they appear; or their names.

If we only have a small number of formats to account for we can get by with two or three different transformations, one to load each format. But it may not be that simple.

If the number of formats is too large, or if there's no way to determine which format to load based on some criteria (filename, source, etc.), we need something smarter. And this is where metadata injection comes in.

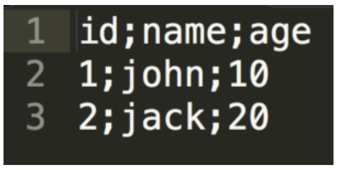
PDI Metadata Injection

PDI has supported Metadata Injection for a few years now. The basic idea consists of having a transformation parse metadata information in runtime, which may vary depending on the circumstances, and then injecting this metadata into the configuration of steps sitting in a separate "template" transformation. Depending on what the data looks like, the template transformation is injected with different values, changing parts of its behaviour. Here's a video to show you how it works, voiced by none other than Matt Casters himself.

Your mileage may vary, but PDI Metadata Injection is a very powerful ally when it comes to creating more generic algorithms.

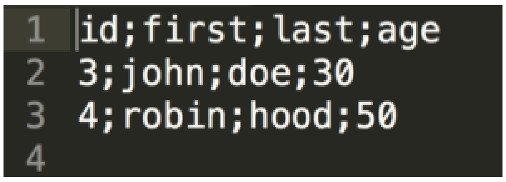
- missing fields (we'll need to add nulls for those)
- extra fields (some fields are not required and should be ignored)
- fields in a different order
- must account for future change
- must allow for easy maintenance

Here's a sample of the files we need to load:



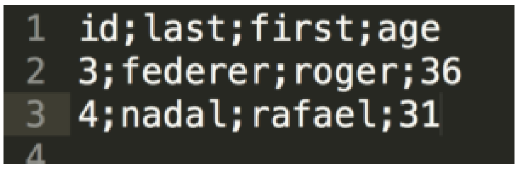
```
id;name;age
1;john;10
2;jack;20
```

Some files have only 1 name field



```
id;first;last;age
3;john;doe;30
4;robin;hood;50
```

Some have 2



```
id;last;first;age
3;federer;roger;36
4;nadal;rafael;31
```

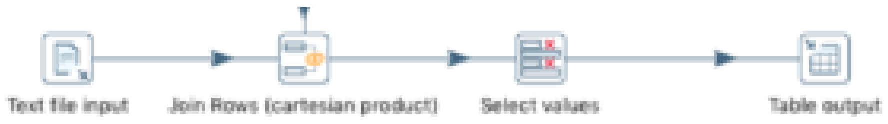
Sometimes they're in a different order

Our goal is to parse all three files using a single PDI transformation with Metadata Injection, allow all configurations to come from an external data dictionary and support future changes to file formats as much as possible.

The template transformation

The template transformation is quite simple:

- a text file input step to read the CSV file;
- a data grid step to add the fields that are missing from the input file
- a select values step to re-order the fields to a fixed output order



Remark: not all PDI steps support Metadata Injection. In particular, the Add Constants and Generate Rows steps don't have support for it, so we can't use them to add the missing fields. The Data Grid step, however, supports Metadata Inection and can therefore be used, followed by a Join Rows, to add extra fields, which can vary from one file to the next. Also, the Select Values step isn't really necessary, as we can inject the list of output fields directly in the Table Output step. But this way we can also attach a Write to Log step right before the Table Output, if we need to debug our transformation.

How does it work?

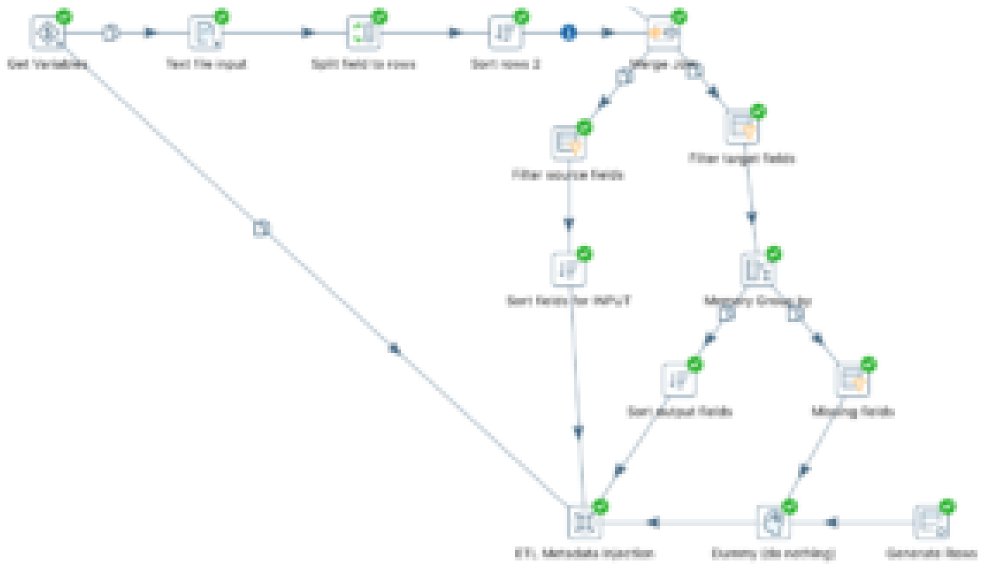
The Text file input step will list all fields that exist in the file, with the desired destination names; the Data grid step will include whatever fields are missing from the input file (e.g., there's no last name field in the first input file) and add a single row of data with value null; once joined to the main data flow, the select values step re-orders the fields in the order we want them and finally a table output step will write to the target database table.

If we take our data dictionary to include 4 fields, **id**, **first_name**, **last_name**, and **age**, the Text File Input step must have 3 fields configured, **id**, **first_name**, **age**; the data grid would have a single field, **last_name**, with a single row of data and null as value; the Select values step would then re-order the fields as **id**, **first_name**, **last_name**, **age**.

The injector transformation

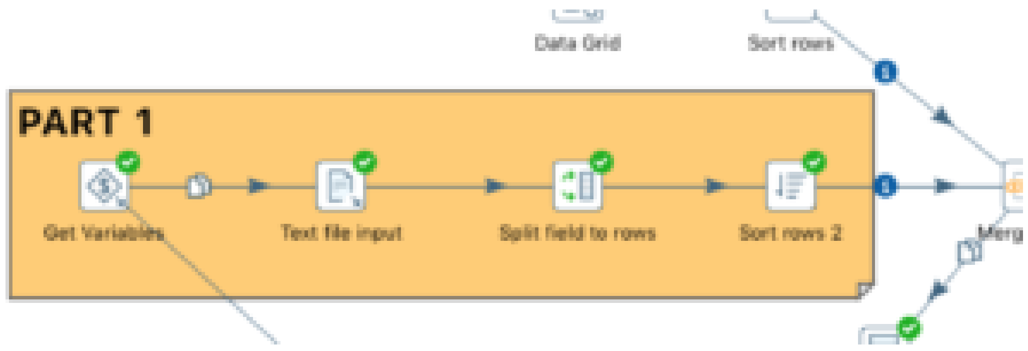
This transformation is the brains of the process. It must determine the file structure, fill in the gaps, and then inject the missing bits in the template transformation so it can load the files one by one.

Here's the structure of the transformation. We'll go over the various steps, one by one, and see what's happening at each stage of the process.



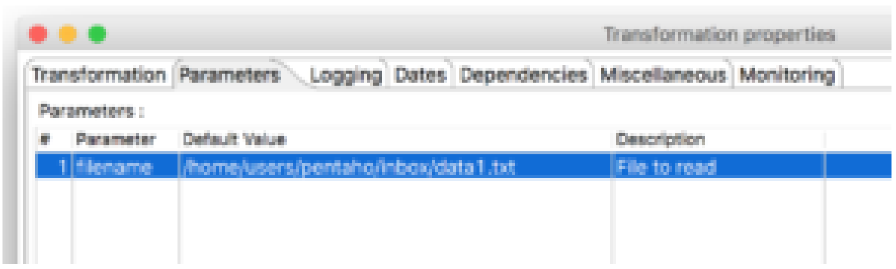
As often happens, injector transformations tend to have multiple streams of data, each carrying part of the set up to the metadata injection step.

Parsing the list of fields



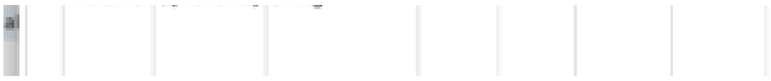
Passing the filename

The filename is first set as a parameter, so that this transformation can then be called by a parent job, once for each of the files waiting to be ingested. The get variables step then retrieves this value.



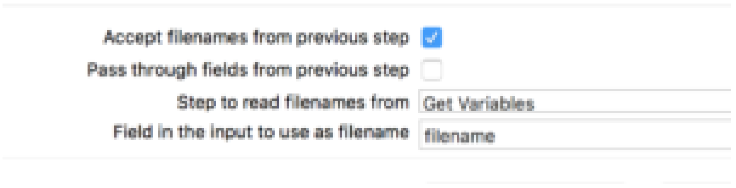
Parameters :

#	Parameter	Default Value	Description
1	filename	/home/users/pentaho/inbox/data1.txt	File to read



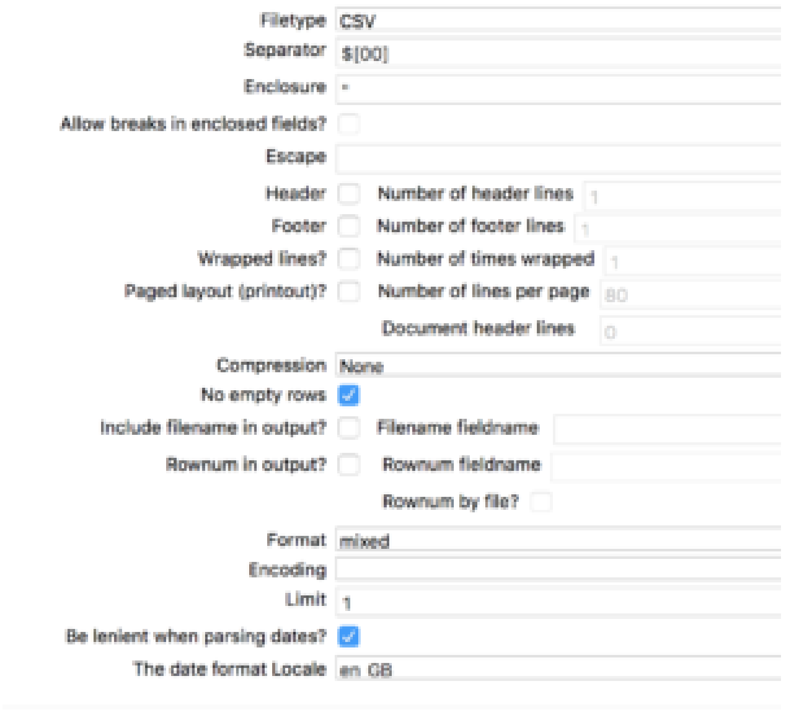
The list of fields

We then read the header row using a Text File Input step.



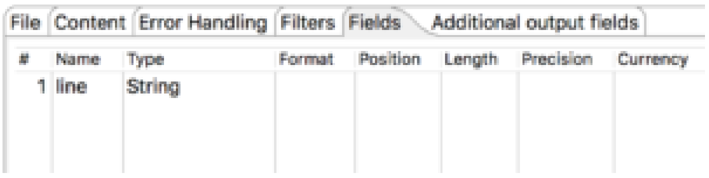
Accept filenames from previous step	✓
Pass through fields from previous step	
Step to read filenames from	Get Variables
Field in the input to use as filename	filename

Reading the filename from the previous step



Filetype	CSV		
Separator	$[00]		
Enclosure	"		
Allow breaks in enclosed fields?			
Escape			
Header		Number of header lines	1
Footer		Number of footer lines	1
Wrapped lines?		Number of times wrapped	1
Paged layout (printout)?		Number of lines per page	80
		Document header lines	0
Compression	None		
No empty rows	✓		
Include filename in output?		Filename fieldname	
Rownum in output?		Rownum fieldname	
		Rownum by file?	
Format	mixed		
Encoding			
Limit	1		
Be lenient when parsing dates?	✓		
The date format Locale	en_GB		

The contents tab



#	Name	Type	Format	Position	Length	Precision	Currency
1	line	String					

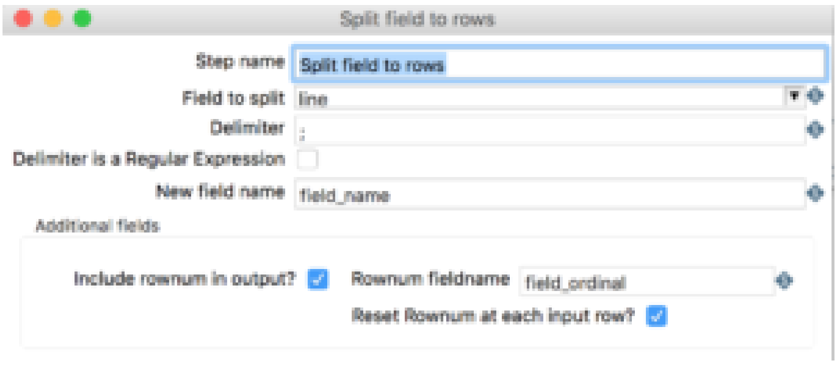
The fields tab

The important bits here:

> We use character **$[00]** as the separator, as it's a character that does not occur naturally in the data. This character is ASCII's character 00, or the null character. It should be safe under most uses, but your mileage may vary.

> As we want to read the header row, we must disable the header option

Next we split use a Split field to Rows step, using semi-colon (the separator in our data files) to split the field:



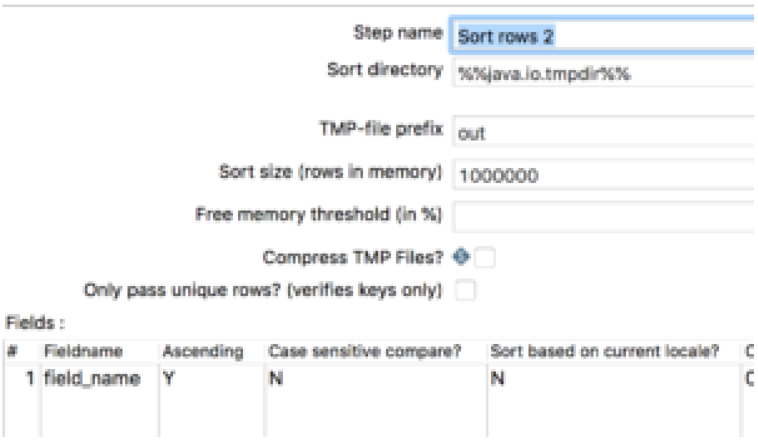
Step name	Split field to rows
Field to split	line
Delimiter	;
Delimiter is a Regular Expression	☐
New field name	field_name

Additional fields

Include rownum in output?	✓	Rownum fieldname	field_ordinal
		Reset Rownum at each input row?	✓

Split field to rows

And we sort the rows of data by field name:



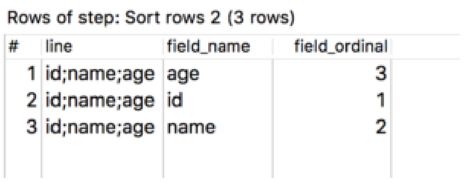
Step name	Sort rows 2
Sort directory	%%java.io.tmpdir%%
TMP-file prefix	out
Sort size (rows in memory)	1000000
Free memory threshold (in %)	
Compress TMP Files?	☐
Only pass unique rows? (verifies keys only)	☐

Fields :

#	Fieldname	Ascending	Case sensitive compare?	Sort based on current locale?	C
1	field_name	Y	N	N	C

Sort fields alphabetically

This is our result. The **field_ordinal** field tells us the order in which the fields occur in the data file.



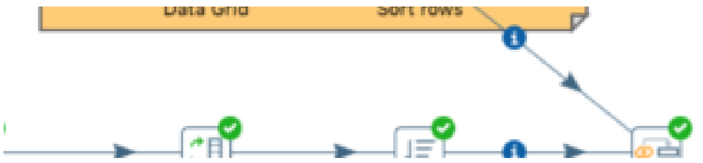
Rows of step: Sort rows 2 (3 rows)

#	line	field_name	field_ordinal
1	id;name;age	age	3
2	id;name;age	id	1
3	id;name;age	name	2

One row per field

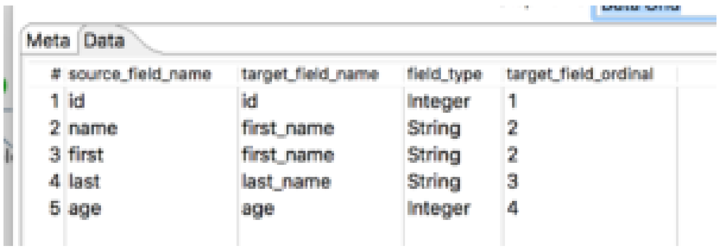
The reason for that sort step will become clear in the next part.

Preparing the data dictionary



We need a data dictionary, where the relationship between data fields coming from the files are mapped to the target fields on the destination database. Here we use a data grid, but on a production ETL this information would be stored in a database.

In this simple example we need the following information:



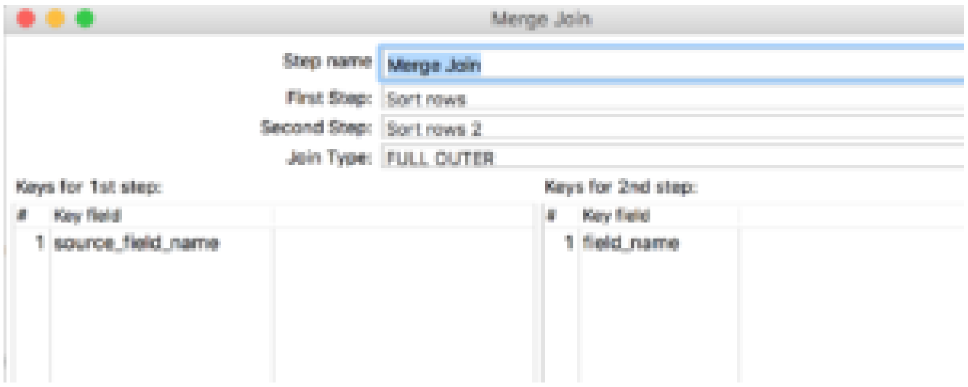
#	source_field_name	target_field_name	field_type	target_field_ordinal
1	id	id	Integer	1
2	name	first_name	String	2
3	first	first_name	String	2
4	last	last_name	String	3
5	age	age	Integer	4

Mapping between source and target field names

We then sort by field name, ahead of joining it back to the field names coming from the file.

Joining the source and target fields

The next step is joining the two streams of data, the field list coming from the data file and the field list coming from the dictionary.



Step name: Merge Join
First Step: Sort rows
Second Step: Sort rows 2
Join Type: FULL OUTER

Keys for 1st step:

#	Key field
1	source_field_name

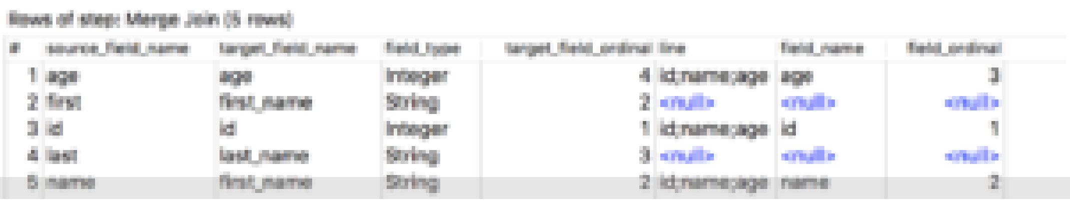
Keys for 2nd step:

#	Key field
1	field_name

The merge join step requires both input streams of data to be sorted by the join keys, which is why we needed to sort both data streams immediately before.

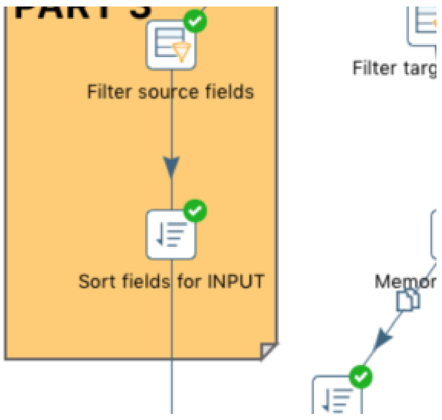
The need for a full outer join will be clear in a while.

This is the resulting data after the join:



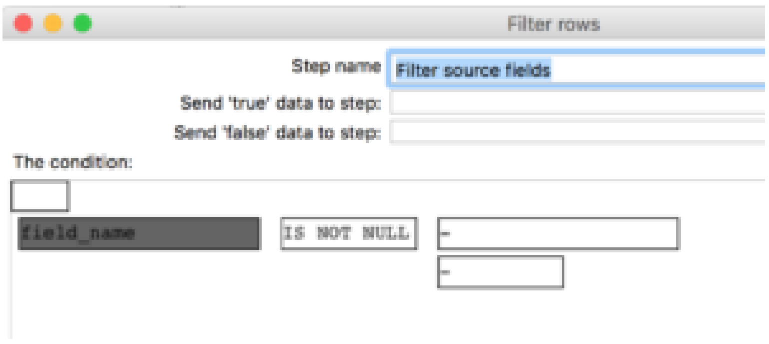
Rows of step: Merge Join (5 rows)

#	source_field_name	target_field_name	field_type	target_field_ordinal	line	field_name	field_ordinal
1	age	age	Integer	4	id;name;age	age	3
2	first	first_name	String	2	<null>	<null>	<null>
3	id	id	Integer	1	id;name;age	id	1
4	last	last_name	String	3	<null>	<null>	<null>
5	name	first_name	String	2	id;name;age	name	2



We now start preparing the various data streams to be injected. The first one is the list of fields to inject in the template's Text File Input step, which will contain all the input fields present in the data file.

A Filter Rows step allows us to select only those rows that correspond to fields present in the data file.

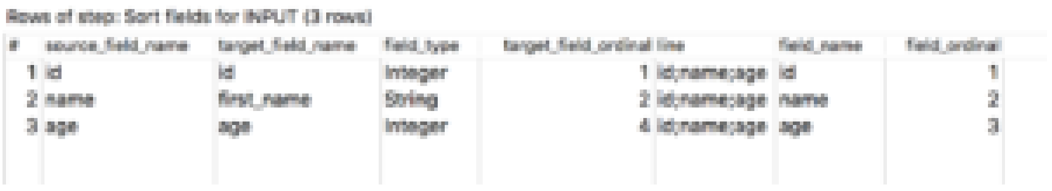

Filter rows

Step name	Filter source fields
Send 'true' data to step:	
Send 'false' data to step:	

The condition:

```
field_name   IS NOT NULL   -
```

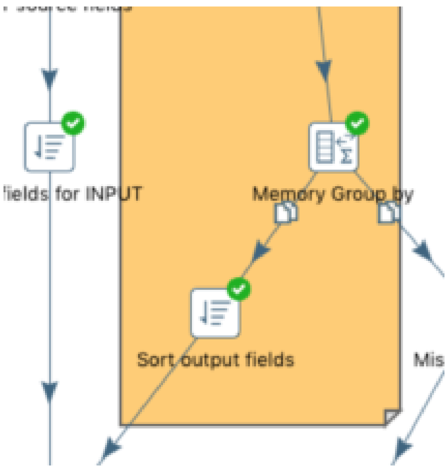
Once sorted by the field_ordinal value (the order in which they appear in the data file, we have:


Rows of step: Sort fields for INPUT (3 rows)

#	source_field_name	target_field_name	field_type	target_field_ordinal	line	field_name	field_ordinal
1	id	id	Integer	1	id;name;age	id	1
2	name	first_name	String	2	id;name;age	name	2
3	age	age	Integer	4	id;name;age	age	3

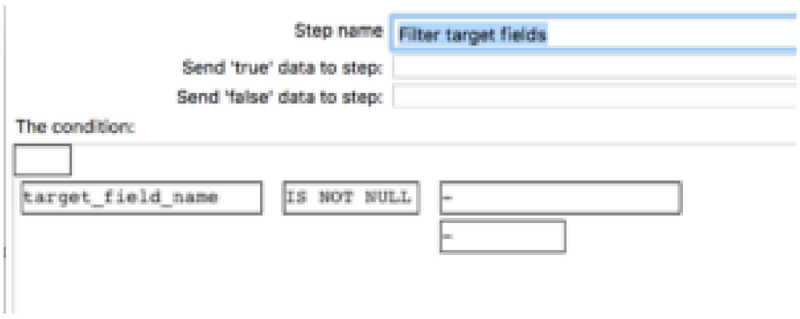
This list of fields exactly matches the order in which they occur in the data file.

The list of target fields



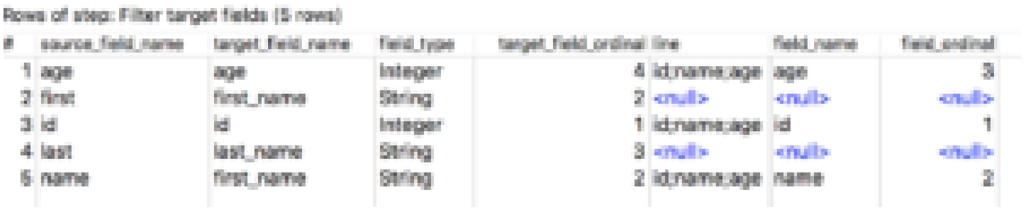
The next step is to prepare the list of target fields. These will be injected in the Select Values and Table output steps.

We start by filtering only those fields that have a match in the data dictionary:


Step name: Filter target fields
Send 'true' data to step:
Send 'false' data to step:
The condition:

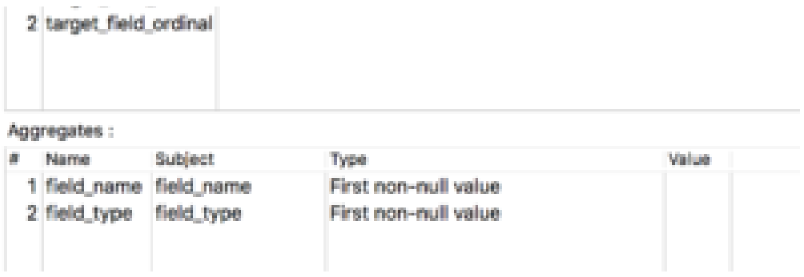
```
target_field_name   IS NOT NULL   -
```

The output of this step will include all fields that found a match in the data dictionary:


Rows of step: Filter target fields (5 rows)

#	source_field_name	target_field_name	field_type	target_field_ordinal	line	field_name	field_ordinal
1	age	age	Integer	4	id;name;age	age	3
2	first	first_name	String	2	<null>	<null>	<null>
3	id	id	Integer	1	id;name;age	id	1
4	last	last_name	String	3	<null>	<null>	<null>
5	name	first_name	String	2	id;name;age	name	2

Note that multiple occurrences of **first_name** are displayed in the output. This is because or our full outer join above. Though we need a full outer join to get both the fields present in the input data file as well as those missing, this then causes the duplicates. We get rid of them with a Group by step (actually, a Memory Group by that doesn't require a sort beforehand),



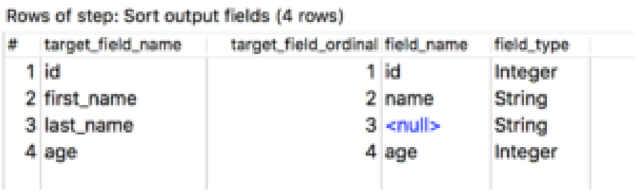
2	target_field_ordinal

Aggregates :

#	Name	Subject	Type	Value
1	field_name	field_name	First non-null value	
2	field_type	field_type	First non-null value	

This way we get a single row of output for each combination **target_field_name-target_field_ordinal**. The Aggregate fields are not used for this section but will be necessary for the next one, where we inject the Data Grid step to add the missing field information.

We finally sort by the **target_field_ordinal** field, which gives us

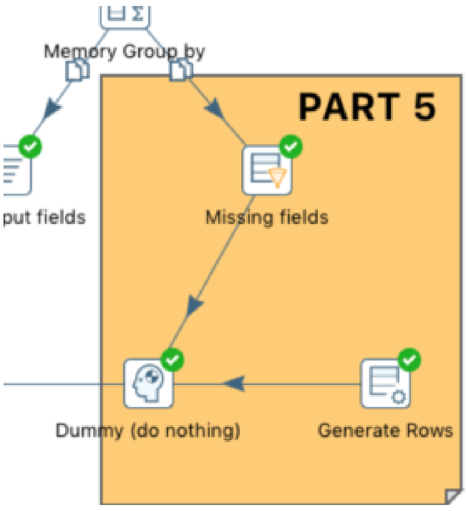


Rows of step: Sort output fields (4 rows)

#	target_field_name	target_field_ordinal	field_name	field_type
1	id	1	id	Integer
2	first_name	2	name	String
3	last_name	3	<null>	String
4	age	4	age	Integer

This is the list of output fields, in the order in which we want them. The order is determined by the ordinal field coming from the data dictionary, so it'll be the same order regardless of which file we're reading.

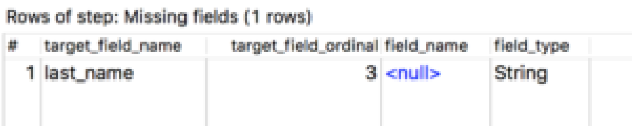
Adding missing fields



In some cases we have files that are missing some of the fields. We can identify those rows, since they will have a null **field_name** after the Group by of the previous section (that's why we chose the "First non-null value" aggregation; **target fields** with non-null **field names** exist in the data file. **Target fields** with only null **field names** do not exist in the data file.

So, we first filter out only those rows for which **field_name** is null:

Which gives this result:



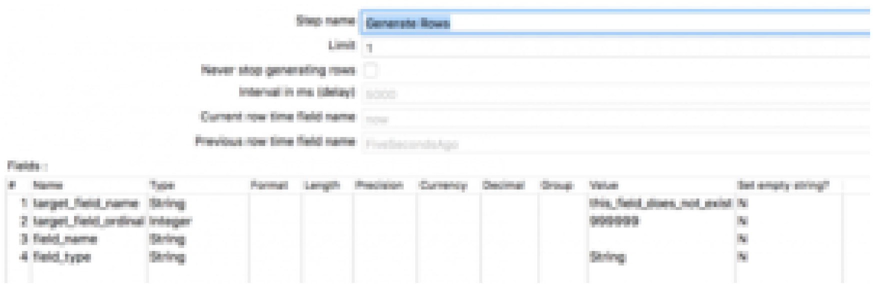
Rows of step: Missing fields (1 rows)

#	target_field_name	target_field_ordinal	field_name	field_type
1	last_name	3	<null>	String

And we'll use this information (only **target_field_name** and **field_type**) to inject into the Data Grid step.

However, since PDI 7.1, if nothing is injected we get an error. If there's nothing to inject, then Data Grid returns 0 rows. When we then cross-join them in the injected transformation the result is 0 rows of data. This behaviour is not observed in a non-injected data grid: even an empty Data Grid step will always output 1 row of data (with no fields and no values). But on Metadata Injection it will output 0 rows.

Because of this we need to always inject at least one row of data in the Data Grid. That's what the Generate Rows step is doing:



Step name: Generate Rows
Limit: 1
Never stop generating rows: ☐
Interval in ms (delay): 5000
Current row time field name: now
Previous row time field name: FiveSecondsAgo

Fields :

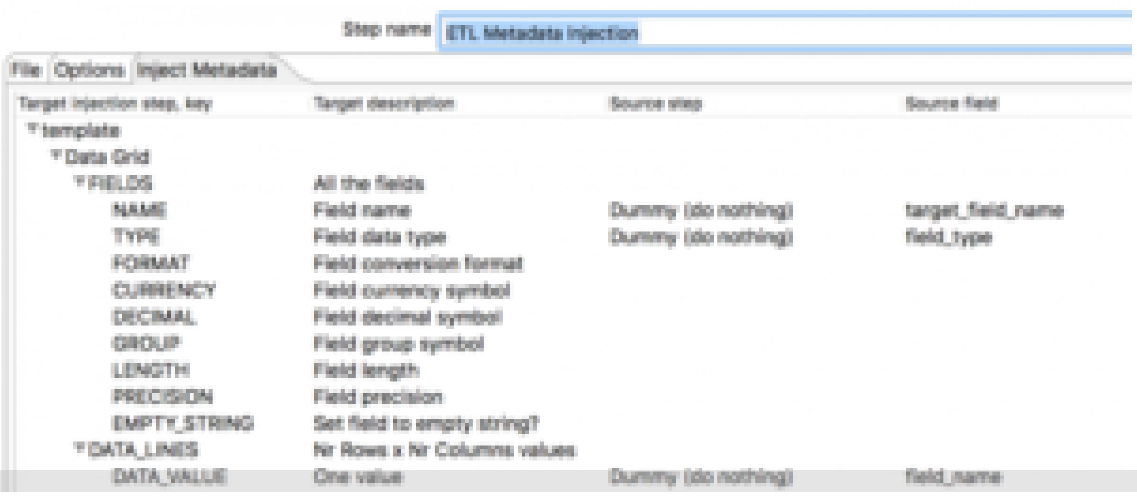
#	Name	Type	Format	Length	Precision	Currency	Decimal	Group	Value	Set empty string?
1	target_field_name	String							this_field_does_not_exist	N
2	target_field_ordinal	Integer							999999	N
3	field_name	String								N
4	field_type	String							String	N

With this fix we ensure there's always at least 1 row of data arriving at the Metadata Injection step from this data stream.

Metadata Injection

At this point we have 3 incoming data streams. We add a new one, directly from the Get Variables step (this will inject the filename), and we're ready to go.

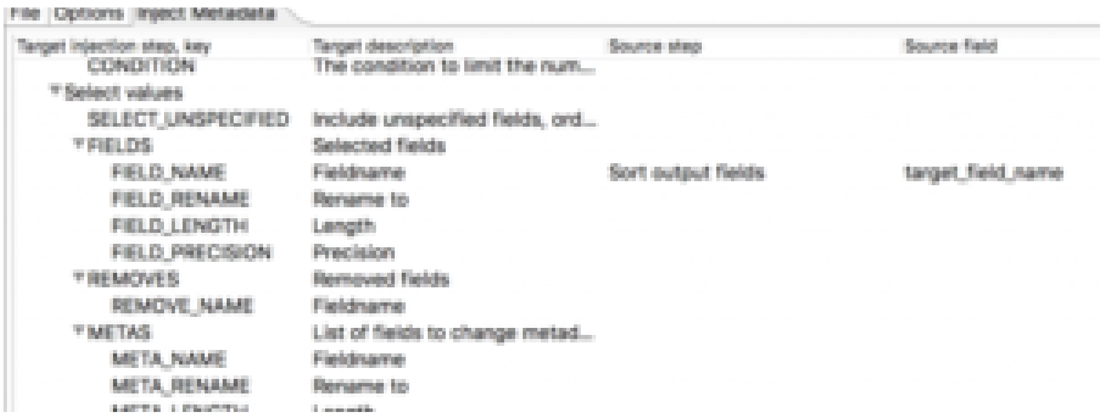
Here is the configuragion of the Metadata Injection step:



Step name: ETL Metadata Injection

File | Options | Inject Metadata

Target injection step, key	Target description	Source step	Source field
template			
Data Grid			
FIELDS	All the fields		
NAME	Field name	Dummy (do nothing)	target_field_name
TYPE	Field data type	Dummy (do nothing)	field_type
FORMAT	Field conversion format		
CURRENCY	Field currency symbol		
DECIMAL	Field decimal symbol		
GROUP	Field group symbol		
LENGTH	Field length		
PRECISION	Field precision		
EMPTY_STRING	Set field to empty string?		
DATA_LINES	Nr Rows x Nr Columns values		
DATA_VALUE	One value	Dummy (do nothing)	field_name

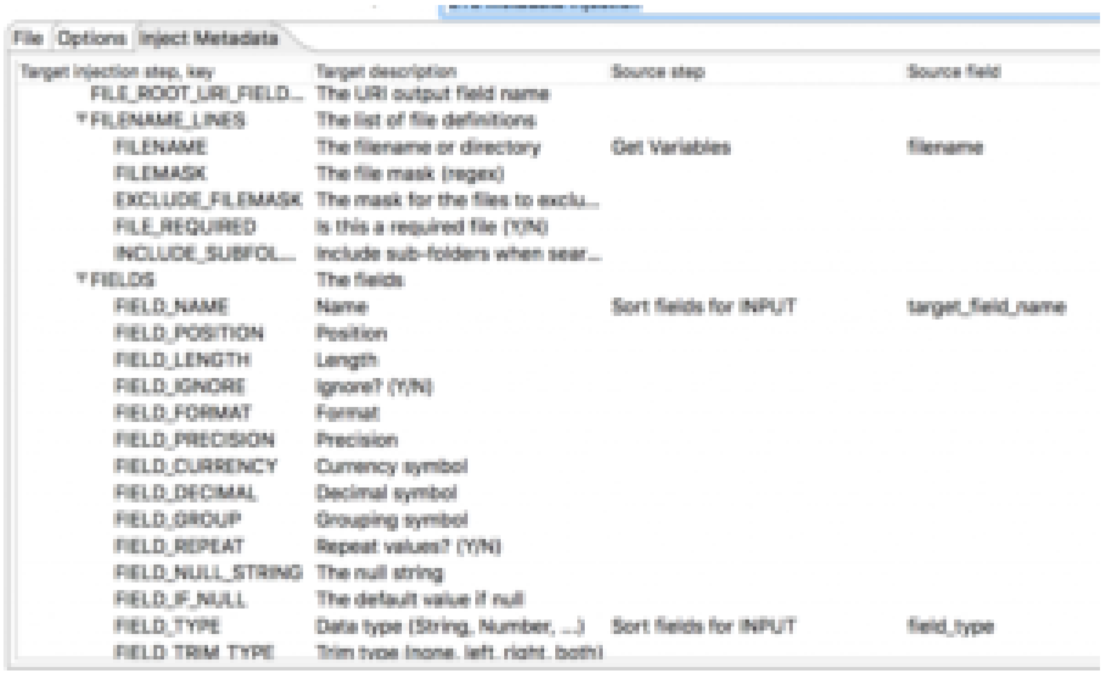


File | Options | Inject Metadata

Target injection step, key	Target description	Source step	Source field
CONDITION	The condition to limit the num...		
▾ Select values			
SELECT_UNSPECIFIED	Include unspecified fields, ord...		
▾ FIELDS	Selected fields		
FIELD_NAME	Fieldname	Sort output fields	target_field_name
FIELD_RENAME	Rename to		
FIELD_LENGTH	Length		
FIELD_PRECISION	Precision		
▾ REMOVES	Removed fields		
REMOVE_NAME	Fieldname		
▾ METAS	List of fields to change metad...		
META_NAME	Fieldname		
META_RENAME	Rename to		

Metadata injection: Select Values

In the Select Values step we inject all the **target_field_names** coming from above, regardless of them existing in the data file or not (as missing fields are added in the Data Grid step).



File | Options | Inject Metadata

Target injection step, key	Target description	Source step	Source field
FILE_ROOT_URI_FIELD...	The URI output field name		
▾ FILENAME_LINES	The list of file definitions		
FILENAME	The filename or directory	Get Variables	filename
FILEMASK	The file mask (regex)		
EXCLUDE_FILEMASK	The mask for the files to exclu...		
FILE_REQUIRED	Is this a required file (Y/N)		
INCLUDE_SUBFOL...	Include sub-folders when sear...		
▾ FIELDS	The fields		
FIELD_NAME	Name	Sort fields for INPUT	target_field_name
FIELD_POSITION	Position		
FIELD_LENGTH	Length		
FIELD_IGNORE	Ignore? (Y/N)		
FIELD_FORMAT	Format		
FIELD_PRECISION	Precision		
FIELD_CURRENCY	Currency symbol		
FIELD_DECIMAL	Decimal symbol		
FIELD_GROUP	Grouping symbol		
FIELD_REPEAT	Repeat values? (Y/N)		
FIELD_NULL_STRING	The null string		
FIELD_IF_NULL	The default value if null		
FIELD_TYPE	Data type (String, Number, ...)	Sort fields for INPUT	field_type
FIELD TRIM TYPE	Trim type (none, left, right, both)		

Metadata injection: Text File Input

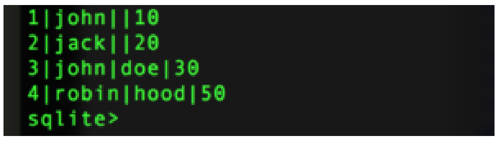
On the text file input step we inject the filename coming from the Get Variables step right at the beginning, and the field name and data type coming from the dictionary (we use the **target_field_name** directly and that way avoid renaming the field later).

And that's it (almost!). Lets test it.

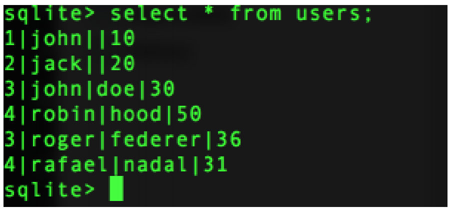
Running the transformation

We first set the filename parameter to point to our first file, **data1.txt**, and run the transformation. Once successful, we can go and see the data on the target database:

```
sqlite> select * from users;
```


```
1|john||10
2|jack||20
3|john|doe|30
4|robin|hood|50
sqlite>
```

And finally, after changing it again to point to file **data2a.txt**, we have:


```
sqlite> select * from users;
1|john||10
2|jack||20
3|john|doe|30
4|robin|hood|50
3|roger|federer|36
4|rafael|nadal|31
sqlite>
```

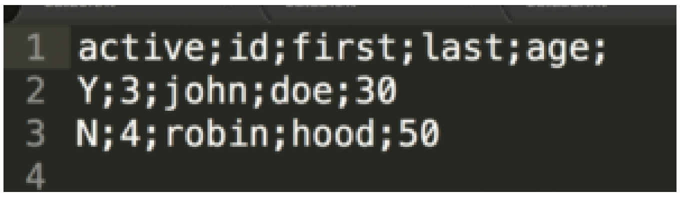
All 3 files loaded correctly, even though they had different structures of data. Our work is done.

Future proofing

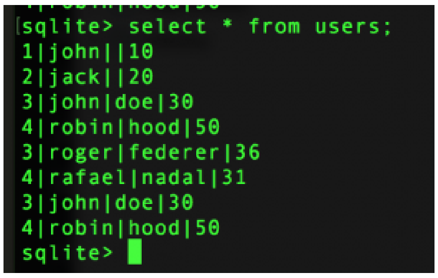
Or is it?

We should test whether our transformation can cope with new fields showing up in the files; fields that the data dictionary isn't yet aware of.

So, let's load one more file, this time with an extra field (and to make it interesting, we'll just add the new field in the beginning):


```
active;id;first;last;age;
Y;3;john;doe;30
N;4;robin;hood;50
```

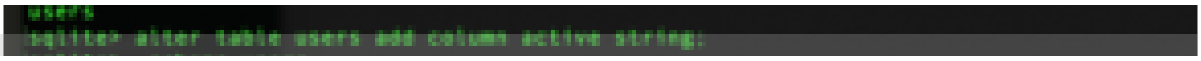
If we switch the transformation to the new file, it'll run successfully and this is what the data shows:

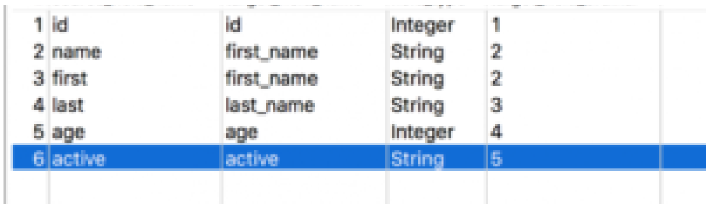

```
sqlite> select * from users;
1|john||10
2|jack||20
3|john|doe|30
4|robin|hood|50
3|roger|federer|36
4|rafael|nadal|31
3|john|doe|30
4|robin|hood|50
sqlite>
```

Indeed, two more rows of data were inserted, and the right fields were written to the right columns. Doesn't have the new column, obviously, as the ETL isn't yet aware of it.

In order to load the new column, we need only two things:

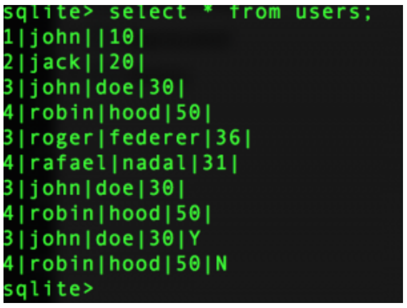
- add a new column to the target table


```
users
sqlite> alter table users add column active string;
```



1	id	id	Integer	1
2	name	first_name	String	2
3	first	first_name	String	2
4	last	last_name	String	3
5	age	age	Integer	4
6	active	active	String	5

Now we can re-run that last file and see the new column populated:



```
sqlite> select * from users;
1|john||10|
2|jack||20|
3|john|doe|30|
4|robin|hood|50|
3|roger|federer|36|
4|rafael|nadal|31|
3|john|doe|30|
4|robin|hood|50|
3|john|doe|30|Y
4|robin|hood|50|N
sqlite>
```

Conclusion

With this transformation we are able to ingest CSV files with variable schema, and add more columns as required. All the data dictionary information can be put in an external database, to make maintenance easier. This allows our ETL to accommodate changes in the data format as they occur.

Of course this is not the end of it. We've only scratched the surface in terms of detecting and parsing the various changes in format. Files can have different separators, encodings, and even date formats, which need to be accounted for. But it's a start, and allows us to, at the cost of spending a bit more time in the initial development, implement an ETL that can, hopefully, survive the test of time.

All this is possible because of PDI's Metadata Injection step. Unfortunately, not all PDI steps can be injected yet so our template algorithm may need some adjustments in order to only injectable steps. However it's a much better position to be in than having to develop different transformations for each possible format and having ETL errors trigger more and more development work.

Download the files

The files used in this article can be downloaded below.

PDI and data files

data.db (SQLite db; put it in your /tmp folder)

Remark for Windows users: to use the transformation as is you need to have SQLite installed, and you'll need to change the database connection in the template transformation to point to a valid path for the data.db file.