Or2yw: Visualize OpenRefine Operations to Workflow Diagram

No Author Given

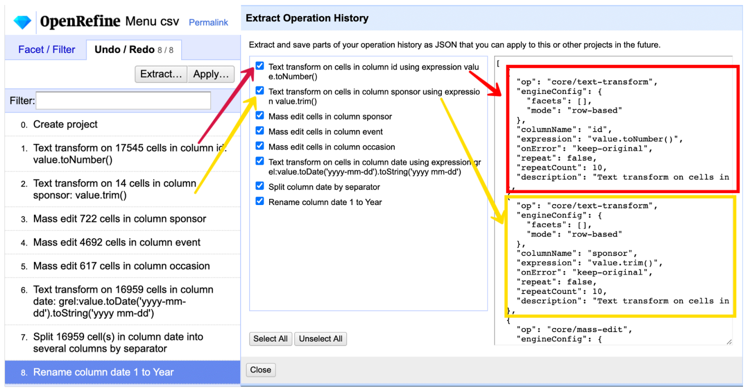
No Institute Given

Abstract. OpenRefine is an open-source data cleaning toolkit, which saves data cleaning transformation operations in JSON formatted text for reusability. While JSON formatted text is mostly machine-readable, they are hard to comprehend in their entirety. We developed a tool that can visualize the OpenRefine operations history into a workflow diagram to make the operations more readable and understandable.
OpenRefine transformation operations are ostensibly linear, but there are two operations levels in OpenRefine, at 1) intra-column and 2) inter-columns. Dependency relationships in data cleaning workflow will change when activities occur at inter-columns. Besides, it is hard to follow lengthy and redundant transformation processes. A high-level abstraction is needed to simplify those actions.
The or2yw toolkit addresses these problems by automatically transforming the OpenRefine operations metadata into a structuralized YesWorkflow model. YesWorkflow is a language-independent tool that can visualize a workflow based on the predefined annotation. This process allows or2yw to visualize each transformation, making it readable and providing implicit operation provenance information at both intra-column and inter-columns in OpenRefine.

Keywords: Data Cleaning · Provenance · Workflow · OpenRefine · Dependency Relationship.

1 Introduction

OpenRefine is an open-source data cleaning toolkit that allows users to do data transformations in a browser-based, spreadsheet-like GUI [4]. In addition to the well-defined User Interface (UI), OpenRefine also records the transformation operations used in a project into an operations history metadata (Figure 1) [3]. On prospective provenance, OpenRefine records the data transformation name, the sequence of operations execution, and the description for each execution, including how many rows are affected by an operation. For example, from Figure 1, we can see the transformation function *toNumber*() is executed for the cells in column *id* at the first step with operation name *Text transform*, and 17,545 cells changed as the result of this operation. For retrospective provenance, OpenRefine also records the column in which the operation is executed and parameters/arguments used in the execution runtime [7].



0. Create project
1. Text transform on 17545 cells in column id: value.toNumber()
2. Text transform on 14 cells in column sponsor: value.trim()
3. Mass edit 722 cells in column sponsor
4. Mass edit 4692 cells in column event
5. Mass edit 617 cells in column occasion
6. Text transform on 16959 cells in column date: grel:value.toDate('yyyy-mm-dd').toString('yyyy mm-dd')
7. Split 16959 cell(s) in column date into several columns by separator
8. Rename column date 1 to Year

Extract Operation History

Extract and save parts of your operation history as JSON that you can apply to this or other projects in the future.

- ✓ Text transform on cells in column id using expression value.toNumber()
- ✓ Text transform on cells in column sponsor using expression value.trim()
- ✓ Mass edit cells in column sponsor
- ✓ Mass edit cells in column event
- ✓ Mass edit cells in column occasion
- ✓ Text transform on cells in column date using expression grel:value.toDate('yyyy-mm-dd').toString('yyyy mm-dd')
- ✓ Split column date by separator
- ✓ Rename column date 1 to Year

```
[
  {
    "op": "core/text-transform",
    "engineConfig": {
      "facets": [],
      "mode": "row-based"
    },
    "columnName": "id",
    "expression": "value.toNumber()",
    "onError": "keep-original",
    "repeat": false,
    "repeatCount": 10,
    "description": "Text transform on cells in
  },
  {
    "op": "core/text-transform",
    "engineConfig": {
      "facets": [],
      "mode": "row-based"
    },
    "columnName": "sponsor",
    "expression": "value.trim()",
    "onError": "keep-original",
    "repeat": false,
    "repeatCount": 10,
    "description": "Text transform on cells in
  },
  {
    "op": "core/mass-edit",
    "engineConfig": {
```

Fig. 1: Operation History in OpenRefine

Using the OpenRefine history metadata, we can infer two types of operation on the OpenRefine, intra-column, and inter-column transformation [8]. First, the intra-column operation or rigid transformation is an operation that is only affecting the value of a cell after transformation is performed. For example, a *To lowercase* transformation apply to a column will only change cells' values in the respective column. Second, the inter-column operation or geometry transformation is an operation that can change both the table's structure and value. The transformation applied to the inter-column operation can break the original table's data lineage. For example, adding a new column or dropping a column. Depending on how the transformations are executed on OpenRefine, both transformations types can depend on other columns.

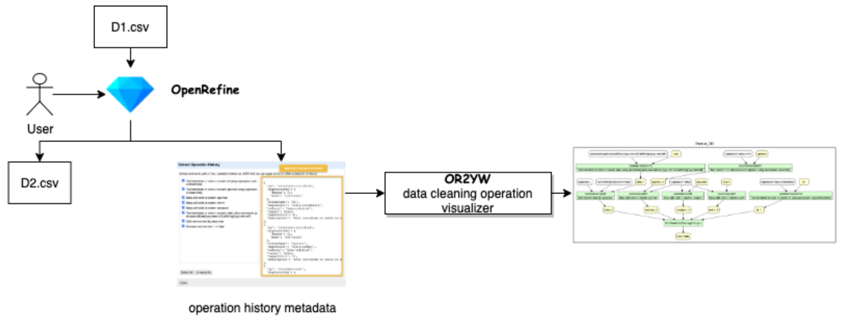

D1.csv
User
OpenRefine
D2.csv
operation history metadata
OR2YW
data cleaning operation visualizer

Fig. 2: a user cleaned a dirty dataset D1.csv file using OpenRefine. This process in OpenRefine will output a clean dataset D2.csv file and an operations history metadata. Or2yw visualizes the operation history into a Yesworkflow diagram.

However, it is not easy to understand each operation's dependencies and effect on the data structure only by looking at the current JSON formatted text metadata. Tracing the lineage from a provenance perspective becomes challenging, especially if there are breakpoints in the processes. Or2yw is developed to help visualizes operation history metadata into a workflow diagram using the Yesworkflow framework [6]. The or2yw visualization tool (Figure 2) helps uncover provenance information by visualizing the operations workflow with its column dependency and effect on the data/table structure.

2 Transformation and Prototype

The or2yw toolkit and the instruction to use the tool are accessible via GitHub [2] and pip [1]. There are three options to visualize workflow in the or2yw, including sequential, parallel, and merge mode.

2.1 Linear

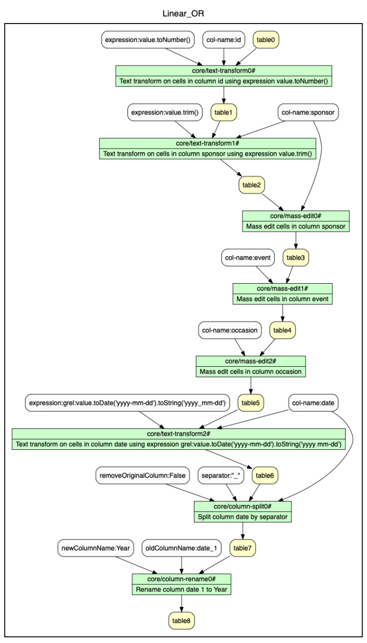

Linear_OR
expression:value.toNumber()
col-name:id
table0
core/text-transform0#
Text transform on cells in column id using expression value.toNumber()
expression:value.trim()
table1
col-name:sponsor
core/text-transform1#
Text transform on cells in column sponsor using expression value.trim()
table2
core/mass-edit0#
Mass edit cells in column sponsor
col-name:event
table3
core/mass-edit1#
Mass edit cells in column event
col-name:occasion
table4
core/mass-edit2#
Mass edit cells in column occasion
expression:grel:value.toDate('yyyy-mm-dd').toString('yyyy_mm-dd')
table5
col-name:date
core/text-transform2#
Text transform on cells in column date using expression grel:value.toDate('yyyy-mm-dd').toString('yyyy mm-dd')
removeOriginalColumn:False
separator:"_"
table6
core/column-split0#
Split column date by separator
newColumnName:Year
oldColumnName:date_1
table7
core/column-rename0#
Rename column date 1 to Year
table8

Fig. 3: Linear: The green box represents transformation, the yellow box represents data stream in the data cleaning workflow, and the white box stands for the input parameters for each transformation

We can see that any transformation operations executed in the OpenRefine are linear by nature from the operation history metadata. This means the order of the execution is important to be able to get the same result. For example, given a transformation T_1 applied on column C_1, if the next transformation T_2 also performed on column C_1, then this following transformation is supposed to be dependent on the effect of T_1. In this situation, the structure of T_2 and T_1 will be sequential. Supposed another transformation T_3 applied to column C_2, the operation history metadata still records T_3 as the dependency of T_2 even though there is no column dependency to the previous transformation. Thus, by default sequence in the operation history is preserving the order of execution $T_1(C_1) \rightarrow T_2(C_1) \rightarrow T_3(C_2)$.

In or2yw, this linear model will visualize workflow as a direct translation of this operation sequence from metadata's operation history. Using the dataset *menu.csv* from The New York Public Library [5], the example in Figure 3 shows the direct translation from the operation history metadata on Figure 1 to a linear workflow. This feature can be considered the basic workflow only based on the execution without additional inference to the column dependency or data structure change.

2.2 Parallel

With the provenance information preserved on the operation history metadata, we can uncover the operation dependency by grouping operations applied to the same column. Besides that, we can also infer the change of structure when an operation introduces a new column. For example, given a transformation T_1 applied on column C_1, and operation T_2 applies to a different column C_2. As the column C_2 is independent of column C_1, the T_2 should be independent with the effect of T_1. Thus, the structure of T_2 and T_1 will be parallel. Supposed there is another operation T_3 performed on the column C_1, the T_3 is dependent to T_1. Thus following this schema, we have two parallel paths $T_1(C_1) \rightarrow T_3(C_1)$ and $T_2(C_2)$.

To visualize these parallel paths in the workflow, or2yw defines three sets of operation nodes according to the type of interventions as can be seen in Figure 4:

Parallel Node: The node in the workflow is parallel to others if it is operating on a different column. Similar to Delpeuch and colleagues work [3], we found that parallel operations can be applied to the operations executed under different columns, as they amount to a pure map on the list of rows. Operations on different columns are expected to be mutually exclusive and not affect each other. In contrast, any actions executed on the same column will follow a sequential procedure. Thus in the workflow output, we can see that the transformation operation can be started from multiple starting points depending on which column is used by that transformation function.

Join Node: When an operation is performed on two or more columns, or2yw presents that operations in a join node. The same rule also applies to an operation that combined columns from preceding parallel nodes. We use the column name from operation history metadata as the signal to determine column dependency. For example, operation T_3 *Add column based on column C1 and column C2*

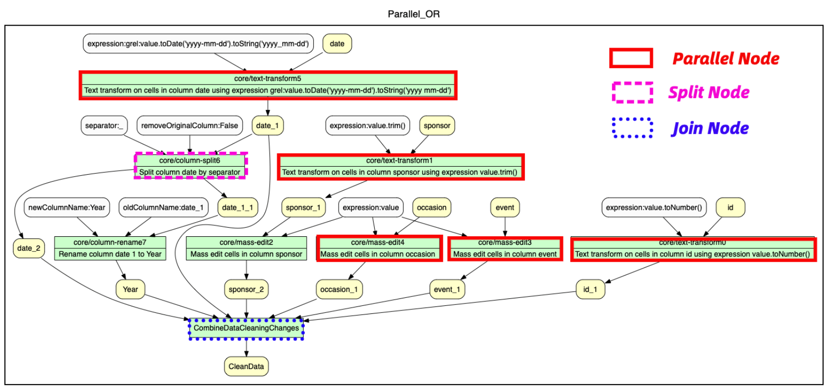

Parallel_OR
Parallel Node
Split Node
Join Node
expression:grel:value.toDate('yyyy-mm-dd').toString('yyyy_mm-dd')
date
core/text-transform5
Text transform on cells in column date using expression grel:value.toDate('yyyy-mm-dd').toString('yyyy mm-dd')
separator:_
removeOriginalColumn:False
date_1
expression:value.trim()
sponsor
core/column-split6
Split column date by separator
core/text-transform1
Text transform on cells in column sponsor using expression value.trim()
newColumnName:Year
oldColumnName:date_1
date_1_1
sponsor_1
expression:value
occasion
event
expression:value.toNumber()
id
date_2
core/column-rename7
Rename column date 1 to Year
core/mass-edit2
Mass edit cells in column sponsor
core/mass-edit4
Mass edit cells in column occasion
core/mass-edit3
Mass edit cells in column event
core/text-transform0
Text transform on cells in column id using expression value.toNumber()
Year
sponsor_2
occasion_1
event_1
id_1
CombineDataCleaningChanges
CleanData

Fig. 4: Parallel: transformations on column *date*, *sponsor*, *occasion*, *event* and *id* are independent; Join Node: input all changed columns at final status to $CombineDataCleaningChanges$, it will return a new table; Split Node: input $date\_1$ to $core/columnsplit6$, two new nodes $date\_1\_1$ and $date\_2$ are generated; .

allows users to create a new column C_3 by concatenating two columns (C_1 and C_2). In this example, T_3 is the join node between C_1 and C_2. Thus, any operation that applies to C_3 will be indirectly depending on C_1 and C_2.

Split Node: a split node in the workflow represents an operation that generates multiple columns. In the OpenRefine, this operation type of operation can be found in split-column transformation. The newly generated columns are mutually exclusive; thus, operations applied to these columns following the generation can be seen as a parallel node. However, similar to the join node, the following operation will be indirectly depending on the split operation's parent column.

2.3 Merge

Merge mode is an additional mode of visualization that allows users to combine the same transformation function that is executed excessively frequent following the linear path. The motivation behind this is to simplify the visualization, so this excessive linear operation will not take so much space on the graphic. In a limited data cleaning study practice, we found this behavior happened when a user performed the merge cluster operation from the clustering operation in the OpenRefine UI. The OpenRefine UI allows users to execute subsequent merge following the previous merging result, thus resulting in multiple mass-cell changes transformation in the operation history metadata.

Or2yw merges these columns by finding the same operations that are executed sub-sequentially on the same column. As a result, these operations will be merged into only one operation node. A separated text file is provided, representing a sub-workflow containing complex operations and storing the parameter information as a node description. Thus, we can visualize a simple workflow for an extensive process with redundant operations.

3 Conclusion

This paper presents or2yw, a toolkit that can visualize the OpenRefine operations to a workflow diagram based on the Yesworkflow framework. The parallel mode in or2yw also helps uncover the relation between operations by presenting each operation dependency based on its input column. Furthermore, the merge mode simplifies the visualization by merging the same subsequent operations in the same column. Our next step to provide better provenance for OpenRefine users is not limited to integrating this workflow tool with OpenRefine but to give a workflow diagram directly from the application or web browser.

Finally, or2yw is not the ultimate solution for providing provenance information. Instead, it is a small step towards data cleaning provenance by visualizing the Data Cleaning workflow in the OpenRefine.

References

1. Anonymous. *pypi for or2yw*, 2020.
2. Anonymous. Visualize openrefine operations to workflow diagram, 2020.
3. Antonin Delpeuch. A complete language for faceted dataflow programs. *arXiv preprint arXiv:1906.05937*, 2019.
4. Lan Li, Bertram Ludäscher, and Qian Zhang. Towards more transparent, reproducible, and reusable data cleaning with openrefine. *iConference 2019 Proceedings*, 2019.
5. The New York Public Library. *Whats on the menu?*, 2020 (10/01/20).
6. Timothy McPhillips, Tianhong Song, Tyler Kolisnik, Steve Aulenbach, Khalid Belhajjame, Kyle Bocinsky, Yang Cao, Fernando Chirigati, Saumen Dey, Juliana Freire, et al. Yesworkflow: a user-oriented, language-independent tool for recovering workflow information from scripts. *arXiv preprint arXiv:1502.02403*, 2015.
7. Leonardo Murta, Vanessa Braganholo, Fernando Chirigati, David Koop, and Juliana Freire. noworkflow: capturing and analyzing provenance of scripts. In *International Provenance and Annotation Workshop*, pages 71–83. Springer, 2014.
8. Santiago Núñez-Corrales, Lan Li, and Bertram Ludäscher. A first-principles algebraic approach to data transformations in data cleaning: Understanding provenance from the ground up. In *12th International Workshop on Theory and Practice of Provenance (TaPP 2020)*, 2020.