Auto-Tables: Relationalize Tables without Using Examples

Peng Li[*]
Georgia Tech
Atlanta, GA
pengli@gatech.edu

Yeye He
Microsoft Research
Redmond, WA
yeyehe@microsoft.com

Cong Yan
Microsoft Research
Redmond, WA
coyan@microsoft.com

Yue Wang
Microsoft Research
Redmond, WA
wanyue@microsoft.com

Surajit Chaudhuri
Microsoft Research
Redmond, WA
surajitc@microsoft.com

ABSTRACT

Relational tables, where each row corresponds to an entity and each column corresponds to an attribute, have been the standard for tables in relational databases. However, such a standard cannot be taken for granted when dealing with tables "in the wild". Our survey of real spreadsheet-tables and web-tables shows that over 30% of such tables do not conform to the relational standard, for which complex table-restructuring transformations are needed before these tables can be queried easily using SQL-based tools. Unfortunately, the required transformations are non-trivial to program, which has become a substantial pain point for technical and non-technical users alike, as evidenced by large numbers of forum questions in places like StackOverflow and Excel/Tableau forums.

We develop an Auto-Tables system that can automatically synthesize pipelines with multi-step transformations (in Python or other languages), to transform non-relational tables into standard relational forms for downstream analytics, obviating the need for users to manually program transformations. We compile an extensive benchmark for this new task, with 244 real test cases collected from user spreadsheets and online forums. Our evaluation suggests that Auto-Tables can successfully synthesize transformations for over 70% of test cases at interactive speeds, without requiring any input from users, making this an effective tool for users to prepare data for downstream analytics.

1. INTRODUCTION

Modern data analytics like SQL and BI are predicated on a standard format of relational tables, where each row corresponds to a distinct "entity", and each column corresponds to an "attribute" for the entities that contains homogeneous data-values. While such tables are de-facto standards in relational databases, to the extent that we as database people may take them for granted, we would like the highlight that a significant fraction of tables "in the wild" actually fail to conform to such standards, making it considerably more challenging to query them using SQL-based tools.

Non-relational tables are common, but hard to query. Real tables in the wild, such as spreadsheet-tables or web-tables, can often be "non-relational" and hard to query, unlike tables that we expect to find in relational databases. We randomly sampled hundreds of user spreadsheets (in Excel), and web tables (from Wikipedia), and found around 30-50% tables to have such issues. Figure 1 and Figure 2 show real samples taken from spreadsheets and the web, respectively, to demonstrate these common issues. (We emphasize that the problem is prevalent at a very large scale, since there are millions of tables like these in spreadsheets and on the web.)

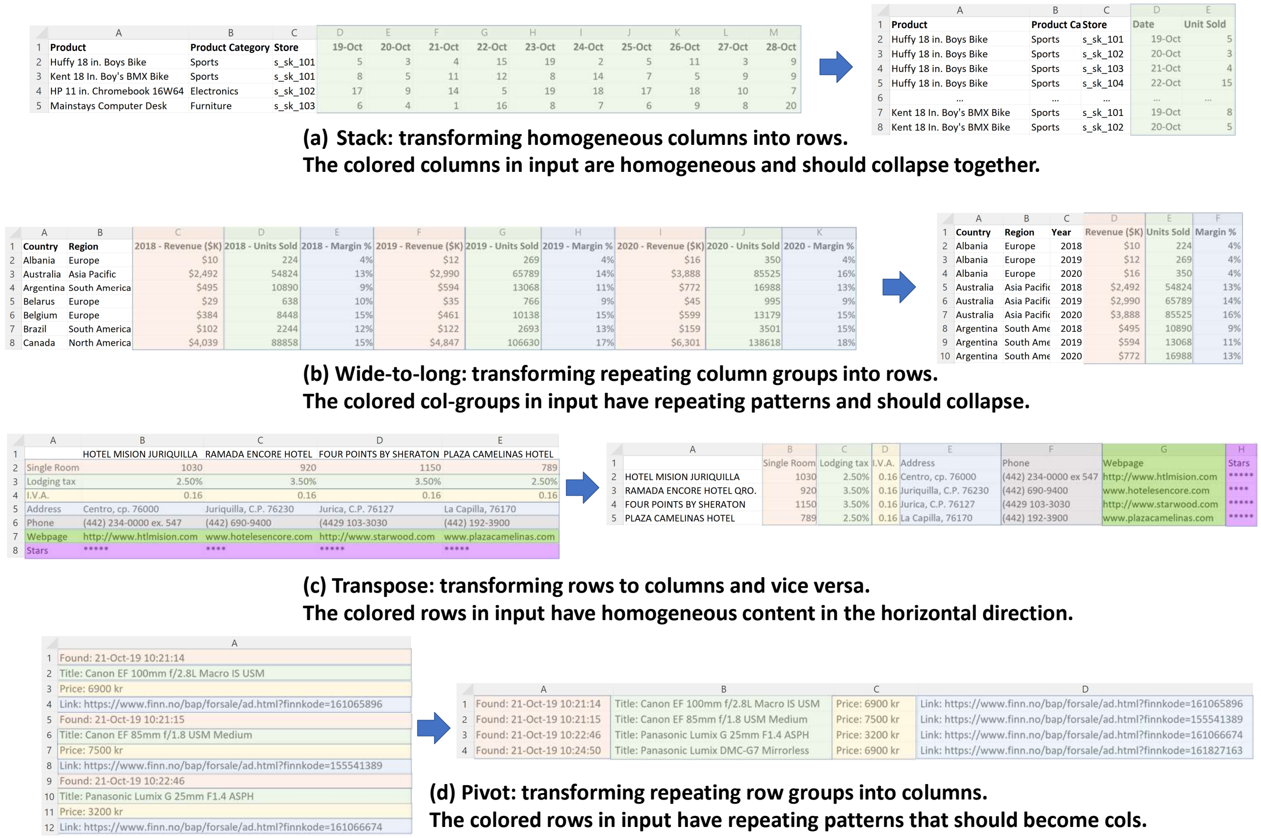
Take Figure 1(a) for example. The table on the left is not a standard relational table, because each column marked in green contains sales numbers for only a single day ("`19-Oct`", "`20-Oct`", etc.), making these column values highly homogeneous in the horizontal direction (while in typical relational tables, we expect values in columns to be homogeneous in the vertical direction). Although this specific table format makes it easy for humans to eyeball changes day-over-day by reading horizontally, it is unfortunately hard to analyze using SQL. Imagine that one needs to compute the 14-day average of sales, starting from "`20-Oct`" – for this table, one has to write: `SELECT SUM("20-Oct", "21-Oct", "22-Oct", ...) FROM T`, across 14 different columns, which is long and unwieldy to write. Now imagine we need 14-day moving averages with every day in October as the starting date – the resulting SQL is highly repetitive and hard to manage.

In contrast, consider a transformed version of this table, shown on the right of Figure 1(a). Here the homogeneous columns in the original table (marked in green) are transformed into only two new columns: "`Date`" and "`Units Sold`", using a transformation operator called "`stack`" (listed in the first row of Table 1). This transformed table contains the same information as the original table, but is much easier to query – e.g., the same 14-day moving average can be computed using a succinct range-predicate on the '`Date`" column, where the starting date "`20-Oct`" is a literal parameter that can be easily changed into other values.

There are many such spreadsheet tables that require different kinds of transformations before they are ready for SQL-based analysis. Figure 1(b) shows another example, where every 3 columns form a repeating group, representing "`Revenue/Units Sold/Margin`" for a different year (marked in red/green/blue in the figure). Tables with these repeat-

[*]Part of work done while at Microsoft.

 This is a minor revision of the paper entitled "Auto-Tables: Synthesizing Multi-Step Transformations to Relationalize Tables without Using Examples", published in PVLDB, Vol. 16, No. 11, 2150-8097. DOI: https://doi.org/10.14778/3611479.3611534
.



(a) Stack: transforming homogeneous columns into rows.
The colored columns in input are homogeneous and should collapse together.

(b) Wide-to-long: transforming repeating column groups into rows.
The colored col-groups in input have repeating patterns and should collapse.

(c) Transpose: transforming rows to columns and vice versa.
The colored rows in input have homogeneous content in the horizontal direction.

(d) Pivot: transforming repeating row groups into columns.
The colored rows in input have repeating patterns that should become cols.

Figure 1: Example input/output tables for 4 operators in Auto-Tables: (a) Stack, (b) Wide-to-long, (c) Transpose, (d) Pivot. The input-tables (on the left) are not relational and hard to query, which need to be transformed to produce corresponding output-tables (on the right) that are relational and easy to query. Observe that the color-coded, repeating row/column-groups are "visual" in nature, motivating a CNN-like architecture like used in computer vision for object-detection.

ing column-groups are also hard to query, just like Figure 1(a), but in this case the required transformation operator is called "`wide-to-long`" (second row of Table 1).

Figure 1(c) shows yet another example, where each hotel corresponds to a column (whose names are in row-1), and each "attribute" of these hotels corresponds to a row. Note that in this case values in the same rows are homogeneous (marked in different colors), unlike relational tables where values in the same columns are homogeneous. A transformation called "`transpose`" is required in this case (listed in the third row of Table 1), to make the resulting table, shown on the right of the figure, easy to query – for instance, a query to sum up the total number of hotel rooms is hard to write on the original table, but can be easily achieved using a simple SUM query on the "`Single Room`" column in the transformed table.

Figure 1(d) shows another example where columns are represented as rows in the table on the left. This is similar to Figure 1(c), except that the rows in this case are "repeating" in groups, thus requiring a different transformation operator called "`pivot`" (listed in the fourth row of Table 1) as opposed to "`transpose`". The resulting table is shown on the right, which becomes easy to query.

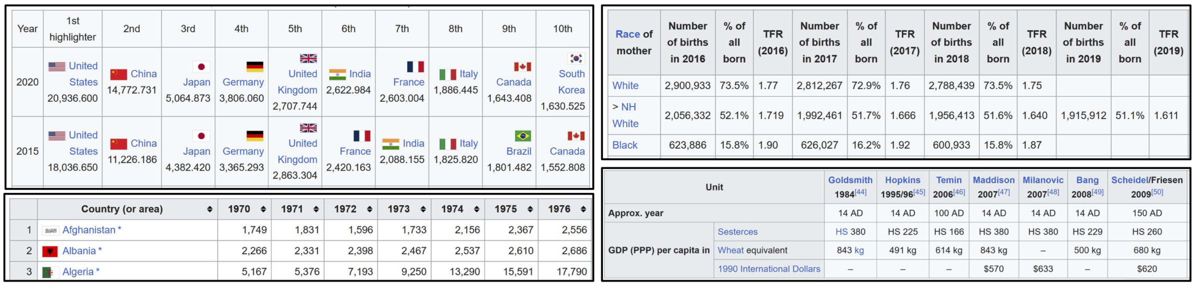
While the examples so far are all taken from spreadsheets, we note that similar structural issues are also widespread in Web tables. Figure 2 shows real examples from Wikipedia, which share similar characteristics as the spreadsheet tables in Figure 1, which all require transformations before these tables can be queried effectively.

Non-relational tables are hard to "relationalize". We mentioned that the example tables in Figure 1 and Figure 2 require different transformation operators. Table 1 shows 8 such transformation operators commonly needed to relationalize tables (where the first 4 operators correspond to the examples we see in Figure 1).

The first column of Table 1 shows the name of the "logical operator", which may be instantiated differently in different languages (e.g., in Python or R), with different names and syntax. The second column of the table shows the equivalent Pandas operator in Python [11], which is a popular API for manipulating tables among developers and data scientists, that readers may be familiar with.

While the operations listed in Table 1 already exist in languages such as R and Python, they are not easy for users to invoke correctly, because users need to:

1. Visually identify different structural issues in an input table that make it hard to query (e.g., repeating row-/column-groups shown in Fig. 1(a-d)), which is not obvious to non-expert users;
2. Map the visual pattern identified from the input table, to a corresponding operator in Table 1 that can handle such issues. This is hard for users not familiar with the exact terminologies to describe these transformations (e.g., `pivot` vs. `stack`);



Year	1st highlighter	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
2020	United States 20,936.600	China 14,772.731	Japan 5,064.873	Germany 3,806.060	United Kingdom 2,707.744	India 2,622.984	France 2,603.004	Italy 1,886.445	Canada 1,643.408	South Korea 1,630.525
2015	United States 18,036.650	China 11,226.186	Japan 4,382.420	Germany 3,365.293	United Kingdom 2,863.304	France 2,420.163	India 2,088.155	Italy 1,825.820	Brazil 1,801.482	Canada 1,552.808

	Country (or area)	1970	1971	1972	1973	1974	1975	1976
1	Afghanistan *	1,749	1,831	1,596	1,733	2,156	2,367	2,556
2	Albania *	2,266	2,331	2,398	2,467	2,537	2,610	2,686
3	Algeria *	5,167	5,376	7,193	9,250	13,290	15,591	17,790

Race of mother	Number of births in 2016	% of all born	TFR (2016)	Number of births in 2017	% of all born	TFR (2017)	Number of births in 2018	% of all born	TFR (2018)	Number of births in 2019	% of all born	TFR (2019)
White	2,900,933	73.5%	1.77	2,812,267	72.9%	1.76	2,788,439	73.5%	1.75			
> NH White	2,056,332	52.1%	1.719	1,992,461	51.7%	1.666	1,956,413	51.6%	1.640	1,915,912	51.1%	1.611
Black	623,886	15.8%	1.90	626,027	16.2%	1.92	600,933	15.8%	1.87			

Unit		Goldsmith 1984[44]	Hopkins 1995/96[45]	Temin 2006[46]	Maddison 2007[47]	Milanovic 2007[48]	Bang 2008[49]	Scheidel/Friesen 2009[50]
Approx. year		14 AD	14 AD	100 AD	14 AD	14 AD	14 AD	150 AD
GDP (PPP) per capita in	Sesterces	HS 380	HS 225	HS 166	HS 380	HS 380	HS 229	HS 260
	Wheat equivalent	843 kg	491 kg	614 kg	843 kg	–	500 kg	680 kg
	1990 International Dollars	–	–	–	$570	$633	–	$620

Figure 2: Real Web tables from Wikipedia that are also non-relational, similar to the spreadsheet tables shown in Figure 1.

Table 1: Auto-Tables DSL: table-restructuring operators and their parameters to "relationalize" tables. These operators are common and exist in many different languages, like Python Pandas and R, sometimes under different names.

DSL operator	Python Pandas equivalent	Operator parameters	Description (example in parenthesis)
stack	melt [14]	start_idx, end_idx	collapse homogeneous cols into rows (Fig. 1a)
wide-to-long	wide_to_long [18]	start_idx, end_idx, delim	collapse repeating col-groups into rows (Fig. 1b)
transpose	transpose [17]	-	convert rows to columns and vice versa (Fig. 1c)
pivot	pivot [15]	repeat_frequency	pivot repeating row-groups into cols (Fig. 1d)
explode	explode [12]	column_idx, delim	convert composite cells into atomic values
ffill	ffill [13]	start_idx, end_idx	fill structurally empty cells in tables
subtitles	copy, ffill, del	column_idx, row_filter	convert table subtitles into a column
none	-	-	no-op, the input table is already relational

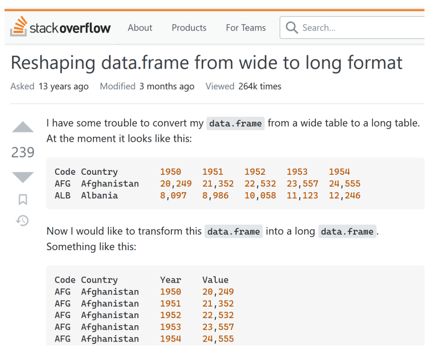


Figure 3: Example user question from StackOverflow, on how to restructure tables. Questions like this are common not only among technical users, but also non-technical users, as similar questions are commonly found on forums for Excel, Power-BI, and Tableau users too [6, 7, 8, 9].

3. Parameterize the chosen operator appropriately (e.g., which columns to collapse, what is the repeating frequency, etc.). This is again hard, as even developers need to consult API documentations that can be long and complex.
4. Certain input tables require more than one transformation step, for which users need to repeat steps (1)-(3).

Completing these steps is a tall order even for technical users, as evidenced by a large number of such questions on forums like StackOverflow. Figure 3 shows one example (popular with many up-votes) – the developer provides example input/output tables to demonstrate the desired transformation, in order to ask what operators should be used.

If technical users like developers find it hard to restructure their tables as these StackOverflow questions would show, it comes as no surprise that non-technical enterprise users, who often deal with tables in spreadsheets, would find the task even more challenging. We find a large number of similar questions on Excel and Tableau forums (e.g., [6, 7, 8, 9]), where users complain that without the required transformations it is hard to analyze data using SQL-based or Excel-based tools (e.g., [2, 3, 4, 5]). The prevalence of these questions confirms table-restructuring as a common pain point for both technical and non-technical users.

Auto-Tables: synthesize transformations without examples. In this work, we propose a new paradigm to automatically synthesize table-restructuring steps to relationalize tables, using the operators in Table 1, *without requiring users to provide examples*. Our key intuition of why we can do away with examples in our task, lies in the observation that given an input table, the logical steps required to relationalize it are *almost always unique*, as the examples in Figure 1 would all show. This is because the transformations required in our task only "restructure" tables, that do not actually "alter" the table content, which is unlike prior work that focuses on *row-to-row transformations* (e.g., TDE [31] and FlashFill [29]), or SQL-by-example (e.g. [22, 50]), where the output is "altered" that can produce many possible outcomes, which would require users to provide input/output examples to demonstrate the desired outcome.

For our task, we believe it is actually important *not* to ask users to provide examples, because in the context of table-to-table transformations like in our case, asking users to provide examples would mean users have to specify *an output table*, which is a substantial amount of typing effort, making it cumbersome to use.

As humans, we can "visually" recognize rows/columns patterns (e.g., homogeneous value groups, as color-coded vertically and horizontally in Figure 1), to correctly predict which operator to use. The question we ask in this paper, is whether an algorithm can "learn" to recognize such patterns by scanning the input tables alone, to predict suitable transformations, in a manner that is analogous to how computer-vision algorithms would scan a picture to identify common but more complex objects like dogs and cats.

In computer vision, in order to pick up subtle clues from pictures, object detection algorithms are typically trained using large amounts of labeled data [26] (e.g., pictures of dogs that are manually labeled as such). In our task, we do not have such labeled datasets. Therefore, we devise a novel *self-training framework* that exploits the *inverse functional relationships* between operators (e.g., the inverse of "`stack`" is known as "`unstack`"), to automatically build large amounts of training data without requiring humans to label, as we will explain in Figure 6. Briefly, in order to build a training example for operator O (e.g., "`stack`"), we start from a relational table R and apply the inverse of O, denoted by O^{-1} (e.g., "`unstack`"), to generate a table $T = O^{-1}(R)$, which we know is non-relational. For our task, given T as input, we know O must be its ground-truth transformation, because by definition $O(T) = O(O^{-1}(R)) = R$, which turns T back to its relational form R. This makes (T, O) an (example, label) pair that we can automatically generate at scale, and use as our training data.

Leveraging training data so generated, we develop an Auto-Tables system that can "learn-to-synthesize" table restructuring transformations, using a deep tabular model we develop inspired by CNN-like architectures popular in the computer vision literature. We show our approach is effective on real-world tasks, which can solve over 70% of test cases collected from user forums and spreadsheets, while being interactive with sub-second latency.

2. RELATED WORK

By-example transformation using program synthesis. There is a large body of prior work on using input/output examples to synthesize transformations. One class of techniques focuses on the so-called "row-to-row" transformations where one input row maps to one output row (e.g., TDE [31] and FlashFill [29]), which are orthogonal to the table-restructuring transformations in Auto-Tables, because these systems do not consider table restructuring operators (Table 1). Other forms of row-to-row transformations using partial specifications (e.g., transform-by-pattern [23, 48], transform-by-target [33, 34], and transform-for-joins [39, 51]), are also orthogonal to the problem we study for the same reason.

A second class of by-example transformation consider "table-to-table" operators, such as Foofah [32] and SQL-by-example techniques like PATSQL [44], QBO [45], and Scythe [46]. These techniques consider a subset of table-restructuring operators, which fall short in the Auto-Tables task as we will show experimentally. It is also worth highlighting, that unlike Auto-Tables that *takes no examples*, these prior systems require users to provide *an example output table*, which is a significant amount of effort required from users.

Computer vision models for object detection. Substantial progress has been made in the computer vision literature on object detection, with variants of CNN architectures being developed to extract salient visual features from pictures [43, 30, 35]. Given the "visual" nature of our problem shown in Figure 1, and the strong parallel between "pixles" in images and "rows/columns" in tables, both of which form two-dimensional rectangles, our model architecture is inspired by CNN-architectures for object detection, but specifically designed for our table transformation task.

3. PRELIMINARY AND PROBLEM

In this section, we will introduce table-restructuring operators, and describe our synthesis problem.

3.1 Table-restructuring operators

We consider 8 table-restructuring operators in our DSL, which are listed in Table 1. Based on our analysis of tables in the wild (in user spreadsheets and on the web), these operators cover a majority of scenarios required to relationalize tables. Note that since our synthesis framework uses self-supervision for training that is not tied to the specific choices of operators, our approach can be easily extended to include additional operators for new functionalities.

In this section, we will introduce the first 4 operators and their parameters shown in Table 1 (we will give additional details in our technical report [1] in the interest of space).

Stack. `Stack` is a Pandas operator [16] (also known as `melt` and `unpivot` in other contexts), that collapses contiguous blocks of homogeneous columns into two new columns. Like shown in Figure 1(a), column headers of the homogeneous columns ("`19-Oct`", "`20-Oct`", etc.) are converted into values of a new column called "`Date`", making it substantially easier to query (e.g., to filter using a range-predicate on "`Date`").

<u>Parameters.</u> In order to properly invoke `stack`, one needs to provide two important parameters, `start_idx` and `end_idx` (listed in the third column of Table 1), which specify the starting and ending column index of the homogeneous column-group that needs to be collapsed. In the case of Figure 1(a), we should use `start_idx`=3 (corresponding to column D) and `end_idx`=12 (column M).

Note that because in Auto-Tables we aim to synthesize complete transformation steps that can execute on input tables, which requires us to predict not only the operators (e.g., `stack` for the table in Figure 1(a)), but also the exact parameters values correctly (e.g., slightly different parameters such as `start_idx`=4 and `end_idx`=12 would fail to produce the desired transformation).

Wide-to-long. `Wide-to-long` is an operator in Pandas [18], that collapses repeating column groups into rows (similar functionality can also be found in R [20]). Figure 1(b) shows such an example, where "`Revenue/Units Sold/Margin`" from different years form column-groups that repeat once every 3 columns. All these repeating column-groups can collapse into 3 columns, with an additional "`Year`" column for year info from the original column headers, as shown on the right in Figure 1(b). Observe that `wide-to-long` is similar in spirit to `stack` (as both collapse homogeneous columns), although `stack` cannot produce the desired outcome when columns are repeating in groups as in this example.

<u>Parameters.</u> `Wide-to-long` has 3 parameters, in which `start_idx` and `end_idx` are similar to the ones used in `stack`. It has an additional parameter called "`delim`", which is the delimitor used to split the original column headers, to produce new column headers and data-values. For example, in the case of Figure 1(b), "`delim`" should be specified as " `-` " to produce: (1) a first part corresponding to values for the new "`Year`" column ("`2018`", "`2019`", etc.); and (2) a second part corresponding to the new column headers in the transformed table ("`Revenue`", "`Units Sold`", etc.).

Transpose. `Transpose` is a table-restructuring operator that converts rows to columns and columns to rows, and is used in other contexts such as in matrix computation. Figure 1(c) shows an example input table on the left, for which `transpose` is needed to produce the output table shown on the right, which would become relational and easy to query.

<u>Parameters.</u> Invoking `transpose` requires no parameters, as all rows and columns will be transposed.

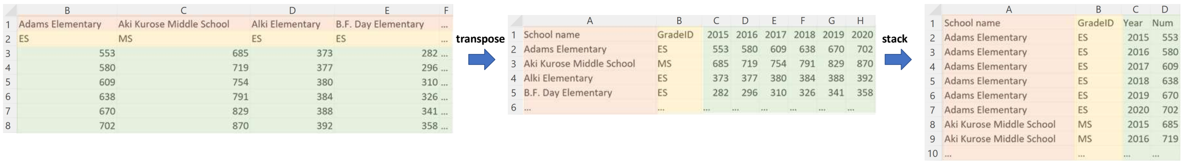


Figure 4: An example input table (on the left) that requires two transformation steps to relationalize: (1) a "**`transpose`**" step to swap rows and columns, (2) a "**`stack`**" step to collapse homogeneous columns (C to H) into two new columns. The resulting output table (on the right) becomes substantially easier to query with SQL (e.g., to filter and aggregate).

Pivot. Like `transpose`, `pivot` also converts rows to columns, as the example in Figure 1(d) shows. However, in this case rows show repeating-groups (whereas in `wide-to-long` columns show repeating-groups), which need to be transformed into columns, like shown on the right of Figure 1(d).

Parameters. `Pivot` has one parameter, "`repeat_frequency`", which specifies the frequency at which the rows repeat in the input table. In the case of Figure 1(d), this parameter should be set to 4, as the color pattern of rows would suggest.

Details of additional operators in Table 1 can be found in our technical report [1].

3.2 Problem statement

Given these table-restructuring operators listed in Table 1, we now introduce our synthesis problem as follows.

Definition 1. Given an input table T, and a set of operators $\mathbf{O} = \{stack, transpose, pivot, \ldots\}$, where each operator $O \in \mathbf{O}$ has a parameter space $P(O)$. Synthesize a sequence of multi-step transformations $M = (O_1(p_1), O_2(p_2), \ldots, O_k(p_k))$, with $O_i \in \mathbf{O}$ and $p_i \in P(O_i)$ for all $i \in [k]$, such that applying each step $O_i(p_i) \in M$ successively on T produces a relationalized version of T.

Note that in our task, we need to predict both the operator O_i and its exact parameters p_i correctly, each step along the way. This is challenging as the search space is large and grows exponentially with the number of steps.

Example 1. Given the input table T shown on the left of Figure 4, the ground-truth transformation M to relationalize T has two-steps: $M =$ (transpose(), stack(start_idx:"2015", end_idx:"2020")). Here the first step "`transpose`" swaps the rows with columns, and the second step "`stack`" collapses the homogeneous columns (between column "`2015`" and "`2020`"). Note that this is the only correct sequence of steps – reordering the two steps, or using slightly different parameters (e.g., start_idx="2016" instead of "2015"), will all lead to incorrect output, which makes the problem challenging.

Also note that although we show synthesized programs using our DSL syntax, the resulting programs can be easily translated into different target languages, such as Python Pandas or R, which can then be directly invoked.

4. AUTO-TABLES: LEARN-TO-SYNTHESIZE

We now describe our proposed Auto-Tables, which learns to synthesize transformations. We will start with an architecture overview, before describing individual components.

4.1 Architecture overview

We represent our overall architecture in Figure 5. The system operates in two modes, with the upper-half of the figure showing the offline training-time pipeline, and the lower-half showing the online inference-time steps.

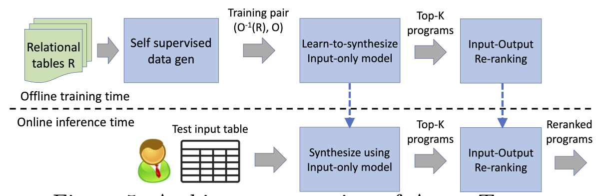


Figure 5: Architecture overview of Auto-Tables

At offline training time, Auto-Tables uses three main components: (1) A "training data generation" component that consumes large collections of relational tables R, to produce (example, label) pairs; (2) An "input-only synthesis" module that learns-to-synthesize using the training data, and (3) An "input-output re-ranking" module that considers both the input table and the output table (produced from the synthesized program), to find the most likely program.

The online inference-time part closely follows the offline steps, where we directly invoke the two models trained offline (the last two blue boxes shown in the figure). Given a user input table, we pass it through our input-only synthesis model, to identify top-k candidate programs, which are then re-ranked by the input-output model for final predictions.

We now describe these three modules in turn below.

4.2 Self-supervised training data generation

As discussed earlier, the examples in Figure 1 demonstrate that there are clear patterns in the input tables that we can exploit (e.g., repeating column-groups and row-groups) to predict required transformations for a given table. Note that these patterns are "visual" in nature, which can likely be captured by computer-vision-like algorithms.

The challenge however, is that unlike computer vision tasks that typically have large amounts of training data (e.g., ImageNet [26]) in the form of (image, label) pairs, in our synthesis task, there is no existing labeled data that we can leverage. Labeling tables manually from scratch are likely too expensive to scale.

Leverage inverse operators. To overcome the lack of data, we propose a novel self-supervision framework leveraging the inverse functional-relationships between operators, to automatically generate large amounts of training data without using humans labels.

Figure 6 shows the overall idea of this approach. For each operator O in our DSL that we want to learn-to-synthesize, we can find its inverse operator (or construct a sequence of steps that are functionally equivalent to its inverse), denoted by O^{-1}. For example, in the figure we can see that the inverse of "`transpose`" is "`transpose`", the inverse of "`stack`" is "`unstack`", while the inverse of "`wide-to-long`" can be constructed as a sequence of 3 steps ("`stack`" followed by "`split`" followed by "`pivot`").

The significance of the inverse operators, is that it allows

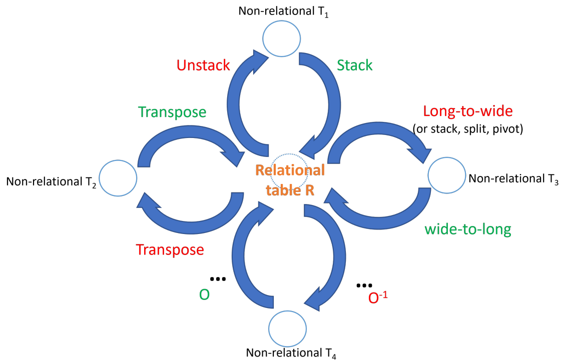


Figure 6: Leverage inverse operators to generate training data. In order to learn-to-synthesize operator O, we can start from any relational table R, apply its inverse operator O^{-1} to obtain $O^{-1}(R)$. Given $T = O^{-1}(R)$ as an input table, we know O must be its ground-truth transformation, because $O(O^{-1}(R)) = R$.

us to automatically generate training examples. Specifically, to build a training example for operator O (e.g., "`stack`"), we can sample any relational table R, and apply the inverse of O, or O^{-1} (e.g., "`unstack`"), to generate a non-relational table $T = O^{-1}(R)$. For our task, given T as input, we know O must be its ground-truth transformation, since by definition $O(T) = O(O^{-1}(R)) = R$, and R is known to be relational. This thus allows us to generate (T, O) as an (example, label) pair, which can be used for training.

Furthermore, we can easily produce such training examples at scale, by sampling: (1) different relational tables R; (2) different operators O; and (3) different parameters associated with each O, therefore addressing our lack of data problem in AUTO-TABLES.

Data Augmentation. Data augmentation [42] is a popular technique to enhance training data and improve model robustness. For example, in computer vision, it is observed that training using additional data generated from randomly flipped/rotated/cropped images, can lead to improved model performance (because an image that contains an object, say dog, should still contain the same object after it is flipped/rotated, etc.) [42].

In the same spirit, we augment each of our relational table R by (1) Cropping, or sampling contiguous blocks of rows and columns in R to produce a new table R'; and (2) Shuffling, or randomly reordering the rows/columns in R to create a new R'. In AUTO-TABLES, we start from over 15K relational tables crawled from public sources [37] (Section 5), and create around 20 augmented tables for each relational table R. This improves data diversity and end-to-end model performance, as we observe in our experiments.

4.3 Input-only Synthesis

After obtaining large amounts of training data in the form of (T, O_p) using self-supervision, we now describe our "input-only" model that takes T as input, to predict a suitable transformation O_p.

4.3.1 Model architecture

We develop a computer-vision inspired model specifically designed for our task, which scans through rows and columns to extract salient tabular features, reminiscent of how computer-vision models extract features from image pixels.

Our model architecture in Figure 7 consists of four sets of layers: (1) table embedding, (2) dimension reduction, (3) feature extraction, and (4) output layers. We will give a brief sketch of each below (details can be found in [1]).

Table embedding layers. Given an input table T, the embedding layer encodes each cell in T into a vector, to obtain an initial representation of T for training. At a high level, for each cell we want to capture both (1) the "*semantic features*" (e.g., people-names vs. company-names), and (2) the "*syntactic feature*" (e.g., data-type, string-length, punctuation, etc.), because both semantic and syntactic features provide valuable signals in our task, e.g., in determining whether rows/columns are homogeneous or similar.

We use the pre-trained Sentence-BERT embedding [40] for semantic features, and encode each cell with 39 pre-defined syntactic attributes (data types, string lengths, punctuation, etc.) as syntactic features, which are then concatenated, as shown in the left half of Figure 7.

Dimension reduction layers. Since the initial representation from the pre-trained Sentence-BERT has a large number of dimensions (with information likely not needed for our task, which can slow down training and increase the risk of over-fitting), we add dimension-reduction layers using two convolution layers with 1×1 kernels, to reduce the dimensionality. Note that we explicitly use 1×1 kernels so that the trained weights are shared across all table-cells, to produce consistent representations after dimension reduction.

Feature extraction layers. We next have feature extraction layers that are reminiscent of CNN [36] but specifically design for our table task. Recall from Figure 1 that the key signals for our task are:

- (1) identify whether values in row or column-directions are "similar" enough to be "homogeneous" (e.g., Figure 1(b) vs. Figure 1(c));
- (2) identify whether entire rows or columns are "similar" enough to show repeating patterns (e.g., Figure 1(b) vs. Figure 1(d)).

Intuitively, if we were to hand-write heuristics, then signal (1) above can be extracted by comparing the representations of adjacent cells in row- and column-directions. On the other hand, signal (2) can be extracted by computing the average representations of each row and column, which can then be used to find repeating patterns.

Based on this intuition, and given the strong parallel between the row/columns in tables and pixels in images, we design feature-extraction layers inspired by *convolution filters* [36] that are popular in CNN architectures to extract visual features from images [35, 43]. Specifically, as shown in Figure 7, we use 1x2 and 1x1 convolution filters followed by average-pooling, in both row- and column-directions, to represent rows/columns/header. Unlike general *nxm* filter used for image tasks (e.g., 3x3 and 5x5 filters in VGG [43] and ResNet [30]), our design of filters are tailored to our table task, because:

- (a) 1x2 filters can easily learn-to-compute signal (1) above (e.g., 1x2 filters with +1/-1 weights can identify the representation differences between neighboring cells, which when averaged, can identify homogeneity in rows/columns).
- (b) 1x1 filters can easily learn-to-compute signal (2) above (e.g., 1x1 filters with +1 weights followed by pooling, correspond to representations of entire rows/columns, which can be used to find repeating patterns in later layers).

We give a more detailed explanation and a concrete example in [1] to illustrate the design here.

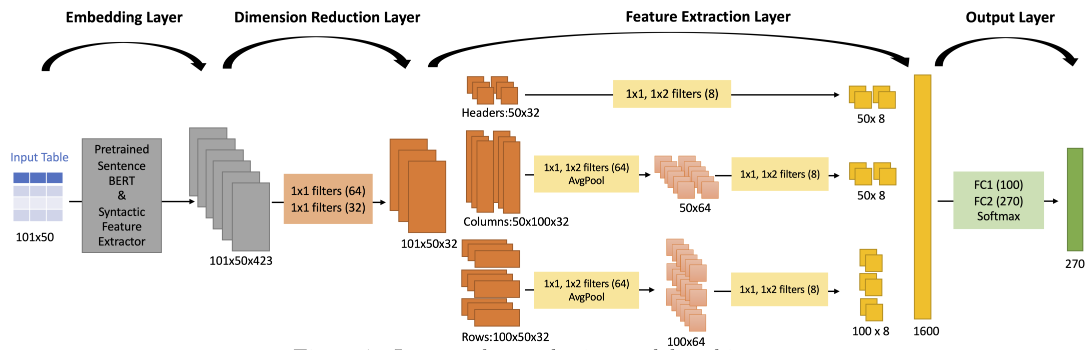


Figure 7: Input-only synthesis: model architecture.

Output layers. Our output layers use two fully connected layers followed by softmax classification, as shown in Figure 7, which produces an output vector that encodes both the predicted operator-type, and its parameters.

4.3.2 Training and inference

We now describe how we train the model, and perform inference to synthesize transformations.

Training time: Loss Function. Given a training input table T, its ground truth operator O and corresponding parameters $P = (p_1, p_2, ...)$, let $\hat{O}$ and $\hat{P} = (\hat{p}_1, \hat{p}_2, ...)$ be the model predicted probability distributions of O and P respectively. The training loss on T can be computed as the sum of loss on all predictions (both the operator-type, and parameters relevant to this operator):

$$Loss(T) = L(O, \hat{O}) + \sum_{p_i \in P, \hat{p}_i \in \hat{P}} L(p_i, \hat{p}_i) \quad (1)$$

Where $L(y, \hat{y})$ denotes the cross-entropy loss [38] commonly used in classification. Given large amounts of training data $\mathbf{T}$ (generated from our self-supervision in Section 4.2), we train our model by minimizing the overall training loss $\sum_{T \in \mathbf{T}} Loss(T)$ using gradient descent until convergence. We will refer to this trained model as H.

Inference time: Synthesizing transformations. At inference time, given an input T, our model H produces a probability for any candidate step O_P that is instantiated with operator O and parameters $P = (p_1, p_2, \ldots)$, denoted by $Pr(O_P|T)$:

$$Pr(O_P|T) = Pr(O) \cdot \prod_{p_i \in P} Pr(p_i) \quad (2)$$

Using the predicted probabilities, finding the most likely transformation step O_P^* given T is then simply:

$$O_P^* = \arg\max_{O,P} Pr(O_P|T) \quad (3)$$

This gives us the most likely one-step transformation given T. As we showed in Figure 4, certain tables may require multiple transformation steps for our task.

To synthesize multi-step transformations, intuitively we can invoke our predictions step-by-step until no suitable transformation can be found. Specifically, given an input table T, at step (1) we can find the most likely transformation O_P^1 for T using Equation (3), such that we can apply O_P^1 on T to produce an output table $O_P^1(T)$. We then iterate, and at step (2) we feed $O_P^1(T)$ as the new input table into our model, to predict the most likely $O_P^2(T)$, and produce an output table $O_P^2(O_P^1(T))$. This iterates until at the k-th step, a "`none`" transformation is predicted (recall that "`none`" is a no-op operator in our DSL in Table 1, to indicate that the input table is already relational and requires no transformations). The resulting $M = (O_P^1, O_P^2, \ldots)$ then becomes the multi-step transformations we synthesize for T.

We defer details of our inference-time algorithm, as well our last component, "input/output re-ranking", to [1].

5. EXPERIMENTS

We perform extensive evaluation on the performance of different algorithms using real test data. Our labeled benchmark data is available on GitHub[1] for future research.

5.1 Experimental Setup

Benchmarks. To study the performance of our method in real-world scenarios, we compile an ATBench benchmark using real cases from three sources:

(1) Online user forums. We sample 23 questions from StackOverflow and Excel user forums, where users ask questions about table restructuring, with sample input/output tables to demonstrate their needs (e.g., Figure 3).

(2) Jupyter notebooks. We sample 79 table-restructuring steps performed by data scientists, extracted from the Jupyter Notebooks (crawled from [47, 48]). We use the transformations programmed by data scientists as the ground truth.

(3) Excel spreadsheets + Web tables. Tables "in the wild" often require transformations before they are fit for analysis (e.g., Figure 1 and 2). We sample 56 such web-tables and 86 spreadsheet-tables, and manually write the desired transformations as the ground truth.

Combining these sources, we get a total of 244 test cases as our ATBench (of which 26 cases require multi-step transformations). Each test case consists of an input table T, ground-truth transformations M_g, and an output table $M_g(T)$ that is relational.

Detailed statistics of the benchmark can be found in [1].

Evaluation Metrics. We evaluate the quality and efficiency of different algorithms in synthesizing transformations.

Quality. Given an input table T, an algorithm A may generate top-k transformations $(\hat{M}_1, \hat{M}_2, ...\hat{M}_k)$, ranked by probabilities, for users to inspect and pick. We evaluate synthesis quality using the standard $Hit@k$ metric [41]:

$$Hit@k(T) = \sum_{i=1}^{k} \mathbf{1}(\hat{M}_i(T) = M_g(T))$$

which looks for exact matches between the top-k ranked predictions $(\hat{M}_i(T), 1 \le i \le k)$ and the ground-truth $M_g(T)$.

[1]`https://github.com/LiPengCS/Auto-Tables-Benchmark`

Table 2: Quality comparison using Hit@k, on 244 test cases

Method	No-example methods				By-example methods			
	Auto-Tables	TaBERT	TURL	GPT-3.5-fs	FF	FR	SQ	SC
Hit @ 1	**0.570**	0.193	0.029	0.196	0.283	0.336	0	0
Hit @ 2	**0.697**	0.455	0.071	-	-	-	0	0
Hit @ 3	**0.75**	0.545	0.109	-	-	-	0	0
Upper-bound	-	-	-	-	0.471	0.545	0.369	0.369

Table 3: Synthesis latency per test case

Method	Auto-Tables	Foofah (excl. 110 timeout cases)	FlashRelate (excl. 91 timeout cases)
50 %tile	**0.127s**	0.287s + human effort	3.4s + human effort
90 %tile	**0.511s**	22.891s + human effort	57.16s + human effort
95 %tile	**0.685s**	39.188s + human effort	348.6s + human effort
Average	**0.224s**	5.996s + human effort	59.194s + human effort

The overall $Hit@k$ on the entire benchmark, is simply the average across all test cases. We report $Hit@k$ up to $k = 3$.

Methods. We experiment using the following methods.

- AUTO-TABLES. This is our approach, and unlike other prior work, is the only one that does not require users to provide input/output examples. To train AUTO-TABLES, We use 15K base relational tables (extracted from Power-BI models crawled from public sources [37]), to generate 1.4M (input-table, transformation) pairs evenly distributed across 8 operators, following the self-supervision procedure in Section 4.2.
- *Foofah (FF)* [32] synthesizes transformations based on input/output examples. We use 100 output cells from the ground-truth output table for Foofah to synthesize programs, using the authors original implementation [10].
- *Flash-Relate (FR)* [24] is another approach to synthesize table-level transformations by examples. We also use 100 example output cells from the ground-truth to synthesize transformations, using an open-source re-implementation of FlashRelate [21].
- *SQLSynthesizer (SQ)* [50] is a SQL-by-example algorithm that synthesizes SQL queries based on input/output examples. We use the authors implementation [22], provide it with 100 example output cells.
- *Scythe (SC)* [46] is another SQL-by-example method. We used the author's implementation [19] and provide it with 100 example output cells, like previous methods.
- *TaBERT* [49] is a pre-trained table representation. we replace the table representation in AUTO-TABLES (i.e., output of the feature extraction layer in Figure 7) with TaBERT's representation, and train the following fully connected layers using the same training data as ours.
- *TURL* [27] is another table representation approach for data integration tasks. Similar to *TaBERT*, we test the effectiveness of TURL by replacing AUTO-TABLES representation with TURL's.
- *GPT* [25] We perform few-shot in-context learning on GPT-3.5 ("gpt-3.5-turbo", accessed in July 2023) as a baseline. We provide one example per operator, for a total of 7 examples in our few-shot prompt.

5.2 Experiment Results

Quality Comparison. Table 2 shows the comparison between AUTO-TABLES and baselines, evaluated on our benchmark with 244 test cases. We group all methods into two classes: (1) "No-example methods" that do not require users to provide any input/output examples, which include our AUTO-TABLES, and variants of AUTO-TABLES that use TaBERT and TURL for table representations, respectively; and (2) "By-example methods" that include Foofah (FF), FlashRelate (FR), SQLSynthesizer (SQ), and Scythe (SC), all of which are provided with 100 ground truth example cells.

As we can see, AUTO-TABLES significantly outperforms all other methods, successfully transforming 75% of test cases in its top-3, *without needing users to provide any examples*, despite the challenging nature of our tasks. Recall that in our task, even for a single-step transformation, there are thousands of possible operators+parameters to choose from (e.g., a table with 50 columns that requires "`stack`" will have 50x50 = 2,500 possible parameters of start_idx and end_idx) and for two-step transformations, the search space is in the millions (e.g., for "`stack`" alone it is $2500^2 \approx 6M$), which is clearly non-trivial.

Compared to other no-example methods, AUTO-TABLES outperforms TaBERT and TURL respectively by 37.7 and 54.1 percentage point on Hit@1, 20.5 and 64.1 percentage point on Hit@3. This shows the strong benefits for using our proposed table representation and model architecture, which are specifically designed for the table transformation task (Section 4.3).

Compared to by-example methods, the improvement of AUTO-TABLES is similarly strong. Considering the fact that these baselines use 100 output example cells (which users need to manually type), whereas our method uses 0 examples, we argue that AUTO-TABLES is clearly a better fit for the table-restructuring task at hand. Since some of these methods (FF and FR) only return top-1 programs, we also report in the last row their "upper-bound" coverage, based on their DSL (assuming all transformations supported in their DSL can be successfully synthesized).

Additional quality results. We report additional results on quality, such as a breakdown by benchmark sources, and Hit@K in the presence of input tables that are already relational (for which AUTO-TABLES should correctly detect and not over-trigger, by performing no transformations), in our technical report [1].

Running Time. Table 3 compares the average and 50/90/95-th percentile latency, of all methods to synthesize one test case. AUTO-TABLES is interactive with sub-second latency on almost all cases, whose average is 0.224. Foofah and FlashRelate take considerably longer to synthesize, even after we exclude cases that time-out after 30 minutes. This is also not counting the time that users would have to spend typing in output examples for these by-example methods, which we believe make AUTO-TABLES substantially more user-friendly for our transformation task.

We report additional results, such as ablation, sensitivity and error analysis, in our technical report [1].

6. CONCLUSIONS AND FUTURE WORK

We propose a new paradigm to synthesize relationalization transformations without examples, obviating the need for users to provide input/output examples, which is a substantial departure from prior work. Future directions include extending the functionality to a broader set of operators, and exploring the applicability of this technique on other classes of transformations.

Acknowledgments. We thank Dr. Kexin Rong and Dr. Xu Chu for their generous support and valuable feedback, as well as three anonymous VLDB reviewers for their helpful comments on our manuscript.

7. REFERENCES

[1] Auto-Tables: full version. https://arxiv.org/abs/2307.14565.

[2] Example Excel forum question: Hard to query without transformations (Retrieved in 02/2023). https://techcommunity.microsoft.com/t5/excel/power-query-data-cleaning-unpivot-transpose-etc/m-p/2400300.

[3] Example Excel forum question: Hard to query without transformations (Retrieved in 02/2023). https://techcommunity.microsoft.com/t5/excel/unpivot-grouped-data/m-p/3686239.

[4] Example Excel forum question: Hard to query without transformations (Retrieved in 02/2023). https://techcommunity.microsoft.com/t5/excel/unpivot-monthly-data/m-p/1867836.

[5] Example Excel forum question: Table analysis provides unexpected results (Retrieved in 02/2023). https://answers.microsoft.com/en-us/msoffice/forum/all/excel-ideas-feature/c9574cf9-dccc-4356-95d3-07d268e39d82.

[6] Example Excel forum question to relationalize tables: Data restructuring using Excel (Retrieved in 02/2023). https://techcommunity.microsoft.com/t5/excel/data-restructuring-using-excel/m-p/287547.

[7] Example Excel forum question to relationalize tables: Pivot chart 4 columns (Retrieved in 02/2023). https://techcommunity.microsoft.com/t5/excel/pivot-chart-4-columns-set-responses-to-4-questions/m-p/2329880.

[8] Example Excel forum question to relationalize tables: Pivot table issue. (Retrieved in 02/2023). https://techcommunity.microsoft.com/t5/excel/pivot-table-issue/m-p/3015448.

[9] Example Excel forum question to relationalize tables: Transpose data for analysis (Retrieved in 02/2023). https://techcommunity.microsoft.com/t5/excel/transposing-data-for-better-analysis/m-p/1297106.

[10] Foofah code on GitHub. https://github.com/umich-dbgroup/foofah.

[11] Pandas API in Python. https://pandas.pydata.org/.

[12] Pandas operator: Explode. (Retrieved in 02/2023). https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.explode.html.

[13] Pandas operator: FFill. (Retrieved in 02/2023). https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.ffill.html.

[14] Pandas operator: Melt. (Retrieved in 02/2023). https://pandas.pydata.org/docs/reference/api/pandas.melt.html.

[15] Pandas operator: Pivot. (Retrieved in 02/2023). https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.pivot.html.

[16] Pandas operator: Stack. (Retrieved in 02/2023). https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.stack.html.

[17] Pandas operator: Transpose. (Retrieved in 02/2023). https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.transpose.html.

[18] Pandas operator: Wide-to-long. (Retrieved in 02/2023). https://pandas.pydata.org/docs/reference/api/pandas.wide_to_long.html.

[19] PATSQL code on GitHub. https://github.com/NAIST-SE/PATSQL.

[20] R operator: pivot-longer, which is similar to Wide-to-long. (Retrieved in 02/2023). https://tidyr.tidyverse.org/reference/pivot_longer.html.

[21] Reimplementation of FlashRelate code on GitHub. https://github.com/BEE-Synth/Bee/tree/291a824622e36fccfa43461e85be3f836e3f4eff/Eval/Benchmarks/Spreadsheet/flashrelate-01.

[22] Scythe code on GitHub. https://github.com/Mestway/Scythe.

[23] Trifacta: Standardize Using Patterns. (Retrieved in 07/2023). https://docs.trifacta.com/display/DP/Standardize+Using+Patterns.

[24] D. W. Barowy, S. Gulwani, T. Hart, and B. Zorn. Flashrelate: extracting relational data from semi-structured spreadsheets using examples. *ACM SIGPLAN Notices*, 50(6):218–228, 2015.

[25] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[26] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[27] X. Deng, H. Sun, A. Lees, Y. Wu, and C. Yu. Turl: Table understanding through representation learning. *ACM SIGMOD Record*, 51(1):33–40, 2022.

[28] Y. Gao, S. Huang, and A. Parameswaran. Navigating the data lake with datamaran: Automatically extracting structure from log datasets. In *Proceedings of the 2018 International Conference on Management of Data*, pages 943–958, 2018.

[29] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

[30] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[31] Y. He, X. Chu, K. Ganjam, Y. Zheng, V. Narasayya, and S. Chaudhuri. Transform-data-by-example (tde) an extensible search engine for data transformations. *Proceedings of the VLDB Endowment*, 11(10):1165–1177, 2018.

[32] Z. Jin, M. R. Anderson, M. Cafarella, and H. Jagadish. Foofah: Transforming data by example. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 683–698, 2017.

[33] Z. Jin, Y. He, and S. Chauduri. Auto-transform: learning-to-transform by patterns. *Proceedings of the VLDB Endowment*, 13(12):2368–2381, 2020.

[34] M. Koehler, E. Abel, A. Bogatu, C. Civili, L. Mazilu, N. Konstantinou, A. A. Fernandes, J. Keane, L. Libkin, and N. W. Paton. Incorporating data context to cost-effectively automate end-to-end data wrangling. *IEEE Transactions on Big Data*, 7(1):169–186, 2019.

[35] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *CACM*, 60(6):84–90, 2017.

[36] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou. A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE transactions on neural networks and learning systems*, 2021.

[37] Y. Lin, Y. He, and S. Chaudhuri. Auto-bi: Automatically build bi-models leveraging local join prediction and global schema graph. *Proceedings of the VLDB Endowment*, 2023.

[38] K. P. Murphy. *Machine learning: a probabilistic perspective.* MIT press, 2012.

[39] A. D. Nobari and D. Rafiei. Efficiently transforming tables for joinability. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 1649–1661. IEEE, 2022.

[40] N. Reimers and I. Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019.

[41] H. Schütze, C. D. Manning, and P. Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press Cambridge, 2008.

[42] C. Shorten and T. M. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019.

[43] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[44] K. Takenouchi, T. Ishio, J. Okada, and Y. Sakata. Patsql: efficient synthesis of sql queries from example tables with quick inference of projected columns. *arXiv preprint arXiv:2010.05807*, 2020.

[45] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 535–548, 2009.

[46] C. Wang, A. Cheung, and R. Bodik. Synthesizing highly expressive sql queries from input-output examples. In *SIGPLAN*, pages 452–466, 2017.

[47] C. Yan and Y. He. Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1539–1554, 2020.

[48] J. Yang, Y. He, and S. Chaudhuri. Auto-pipeline: synthesizing complex data pipelines by-target using reinforcement learning and search. *Proceedings of the VLDB Endowment*, 2021.

[49] P. Yin, G. Neubig, W.-t. Yih, and S. Riedel. Tabert: Pretraining for joint understanding of textual and tabular data. *arXiv preprint arXiv:2005.08314*, 2020.

[50] S. Zhang and Y. Sun. Automatically synthesizing sql queries from input-output examples. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 224–234. IEEE, 2013.

[51] E. Zhu, Y. He, and S. Chaudhuri. Auto-join: Joining tables by leveraging transformations. *Proceedings of the VLDB Endowment*, 10(10):1034–1045, 2017.