Speeding Up Transformation Search with Lightweight Statistics

Rubab Zahra Sarfraz
rsarf@uic.edu
University of Illinois Chicago

Boris Glavic
bglavic@uic.edu
University of Illinois Chicago

ABSTRACT

The need to construct data transformations arises frequently in data preparation, integration, discovery, and analysis. Systems like Auto-Join [16] or FlashFill [5] automate this task by composing primitive operations into transformations that achieve a given objective. *Transformation search* is challenging due to the large search space as well as the diversity of operations and objectives. Because of this heterogeneity, optimizations proposed in prior work are often limited to specific applications. In this work, we leverage cheap column-level statistics to improve the search efficiency of any transformation search system by pruning a large portion of the search space. Integrating our approach into the Auto-Join system, we improve its runtime by $\sim 2000\times$ on average.

VLDB Workshop Reference Format:
Rubab Zahra Sarfraz and Boris Glavic. Speeding Up Transformation Search with Lightweight Statistics. VLDB 2026 Workshop: Tabular Data Analysis (TaDA).

1 INTRODUCTION

Due to the heterogeneity of data and schemas, transformations are often a necessary precursor of data discovery, integration, preparation, and analysis. Manually composing such transformations from *atomic operations* is a challenging and time-consuming task, and in most cases, requires both domain knowledge as well as a deep understanding of downstream tasks.

However, efficient automation of *transformation search* is challenging for several reasons: (i) *large search space*: most tasks require atomic operations to be composed into sequences or workflows, resulting in a search space that is exponential in the number of steps in a transformation. Furthermore, the number of candidate operations for each step is already quite large as operations are often parameterized. (ii) *diversity of operations:* use cases of transformation search are quite diverse in terms of the operations they consider (e.g., string operations for programming-by-example in spreadsheets versus type and unit conversions for discovery of unionable tables). The net effect is that strategies for pruning the search space are often specific to the set of considered operations and do not generalize to other use cases. (iii) *lack of effective progress metrics:* efficient exploration of the large search space requires heuristics that estimate, for a partial solution, how much progress has been made towards the target to guide search into productive directions. However, for transformation search there are often no obvious ways to estimate progress, requiring the execution of many candidate transformations.

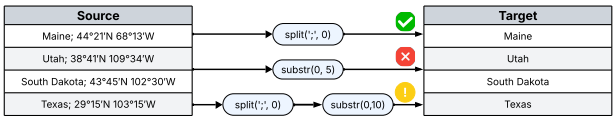

Figure 1: Searching for transformations for equi-join.

These challenges have motivated a broad body of work on automating transformation search across several settings: generalizing input-output examples for spreadsheet transformations into transformation programs [1, 5–7, 14, 17], transformation search for data discovery and joinability [10, 16], interactive data wrangling [8], learning-based program synthesis using large language models [11], and data-driven optimizations that exploit syntactic or semantic profiles to accelerate search [12, 13].

However, despite this progress, transformation search remains an expensive operation, and existing optimizations are mostly limited to specific tasks and classes of transformations. In this work, we propose a radically different approach for improving the efficiency of transformation search. Instead of proposing yet another transformation search method, we identify common primitives that are applied in the majority of transformation search algorithms and study techniques for optimizing these primitives. Any existing transformation search algorithm can then utilize this library to improve search performance. As operations are typically parameterized, e.g., $\mathrm{substr}(x, \mathrm{b}, \mathrm{e})$ returns a substring of x and is parameterized by b (start) and e (end), search algorithms have to apply many candidate operations to columns[1] for a large space of parameter settings. For example, in Figure 1, a valid sequence of operations to transform the left column (C_{source}) into the right column is to apply $\mathrm{split}(C_{source}, \text{`;'}, 0)$ (splitting on semicolon and returning the first element). A transformation search algorithm will have to test many such candidate operations. However, most such candidates will fail, e.g., $\mathrm{split}(C_{source}, \text{`,'}, 0)$ is not applicable as there are no commas in the source column. We model such failure cases through preconditions of operations. We demonstrate how to automatically select appropriate lightweight statistics which we refer to as *column profiles*, and use them to test preconditions of operations without having to evaluate them over all values in a column. Furthermore, we exploit profiles to prune large parts of an operation's parameter space. As long as the preconditions are correct, we only prune candidates guaranteed to fail, preserving the results of the underlying search algorithm. The preconditions can be manually crafted by the user or inferred using existing techniques such as symbolic or concolic execution [2, 3], LLM-based precondition synthesis [9, 15], or program transformation-based predicate formulation [4].

[1]Even though transformation algorithms such as creating spreadsheet transformations by example would seem to not fit this pattern, we can in fact consider the input examples (output examples) as columns to handle such approaches in our framework.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

2 FRAMEWORK

We now present our framework for pruning the search space during transformation search using operation preconditions and column-level statistics. We first describe transformation operations and preconditions supported by our framework, then we cover lifting of operation preconditions to profile-based preconditions, and finally demonstrate how these are leveraged during search.

2.1 Operations and Preconditions

Data model. We consider scalar types including $\mathbb{B}$ (Boolean values), $\mathbb{S}$ (strings), $\mathbb{A}$ (single characters), $\mathbb{N}$ (integers), ...and use $v, v_1, \ldots$ to denote *values* of a scalar type. Every scalar type contains the special value $\perp$. A *column* C is a vector of scalar values of a type τ:

$$C = \langle v_1, v_2, \ldots, v_n \rangle$$

We use $[\tau]$ to denote the type of vectors with element type τ, use $\mathbb{C}, \mathbb{C}', \mathbb{C}_1, \ldots$ to denote column types, $C[i]$ to denote the i^{th} element of a column C, and $|C|$ to denote the number of elements in C.

Atomic operations. We consider transformations that are built from *atomic operations*, or *operations* for short, which are strictly typed functions that take as input one or more arguments of a column type and return one or more columns. Optionally, operations can take additional non-column arguments that we refer to as *parameters*. To simplify notation, we only consider operations that take a single column of type $\mathbb{C}_{in}$ as input (by convention the first argument of the operation) and return a single column of type $\mathbb{C}_{out}$. That is, an operation op with n parameters has type:

$$\text{op}: \underbrace{\mathbb{C}_{in}}_{\textbf{input col.}} \times \underbrace{\tau_1 \times \ldots \times \tau_n}_{\textbf{parameters}} \rightarrow \underbrace{\mathbb{C}_{out}}_{\textbf{output col.}}$$

For example, a parameterless operation tolower that changes every value in a string column to lower case would have type $[\mathbb{S}] \rightarrow [\mathbb{S}]$ and the operation substr with two parameters (begin and end) has type $[\mathbb{S}] \times \mathbb{N} \times \mathbb{N} \rightarrow [\mathbb{S}]$. Note that we expect operations to return $\perp$ to indicate failure.[2] As a convention, scalar arguments of functions are denoted as x, column arguments as X, and parameters as p and other lowercase letters. We often will specify operations as functions that take scalar values as input and implicitly assume that their semantics on columns is defined as mapping the scalar operation over the values in a column. For instance, the substr operation from above, could be defined as a scalar operation with type $\mathbb{S} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{S}$:

$$\text{substr}(x, \text{b}, \text{e}) \coloneqq \begin{cases} x[\text{b} : \text{e}] & \textbf{if } \text{b} < \text{e} \wedge \text{e} < \text{len}(x) \wedge x \neq \perp \\ \perp & \textbf{otherwise} \end{cases}$$

That is, substr returns the substring of v between b and e if (i) b < e, (ii) the string is longer than e characters, and (iii) the input is not $\perp$. Otherwise, the operation fails (returns $\perp$).

Preconditions. It is often possible to identify preconditions for operators that determine whether an operation is applicable to a value in a column (returns $v \neq \perp$ on inputs other than $\perp$). Formally, for an operator op : $[\tau_{in}] \times \tau_1 \times \ldots \times \tau_n \rightarrow [\tau_{out}]$, a precondition $\psi : \tau_{in} \times \tau_1 \times \ldots \times \tau_n \rightarrow \mathbb{B}$ has to satisfy for all $v \neq \perp$:

$$\neg\psi(x, \text{p}_1, \ldots, \text{p}_n) \Rightarrow \text{op}(x, \text{p}_1, \ldots, \text{p}_n) = \perp$$

That is, preconditions are sound indicators of inapplicability of operations: if the precondition fails for a value v and parameter values $v_1, \ldots, v_n$ then the operation will fail on v for this parameter setting. This is important, because for more complex operations, it may be infeasible to specify a condition that covers all possible failure cases. The power and generality of our framework stems from the fact that inapplicability of operations to a column under some parameter settings can be determined based on these preconditions alone without requiring further semantic understanding of the operation, the objective of the transformation search, and how the search algorithm traverses the search space. Furthermore, we present techniques for automatically lifting a general class of operation preconditions to preconditions over column statistics.

For example, consider the substring operation substr from above, one possible precondition is that the input string x should be long enough for the substring to be well-defined:

$$\psi'(x, \text{b}, \text{e}) \coloneqq \text{len}(x) > \text{e} \tag{1}$$

As another example, consider operation $\text{split}(x, \text{d}, \text{j})$ which splits string x on each occurrence of delimiter d and returns the j^{th} (starting from 0) element. Obviously this is only possible if d appears at least j times in x. Using a function $\text{count}(\text{s}, \text{u})$ that counts the number of occurrences of u in string s, we can test this with the following precondition:

$$\psi''(x, \text{d}, \text{j}) \coloneqq \text{count}(x, \text{d}) \geq \text{j} \tag{2}$$

This precondition counts occurrences of a delimiter in a string ensuring that there are sufficiently many occurrences for the split to produce at least j + 1 elements.

Given an application-specific threshold θ on the fraction of values in a column for which an operation has to be applicable to be considered a valid candidate, we can lift a scalar precondition ψ to a column-level condition Ψ_θ:

$$\Psi_\theta(X, \text{p}_1, \ldots, \text{p}_n) \coloneqq \frac{|\{i \mid i \in [1, |X|] \wedge \psi(X[i], \text{p}_1, \ldots, \text{p}_n)\}|}{|X|} \geq \theta$$

The threshold will be dependent on the application and may even vary during search. For instance, for programming-by-example, we may want to require that the operation is applicable to all user inputs (the full column), while for transformation search for join discovery, it may be acceptable if some fraction of column values cannot be transformed and thus will not find join partners.

2.2 Column Profiles and Preconditions

Operation preconditions can be used to prune candidate instantiations of operations for a given input column C and parameter setting. However, this requires evaluating the precondition over the full column. A major insight we exploit in this work is that for every precondition ψ in a very general class of preconditions it is possible to automatically derive a profile-based precondition $\hat{\Psi}$ that implies or even is equivalent to Ψ and can be tested based on lightweight column-level statistics we call *column profiles*. To speed up transformation search, we would generate a column profile containing the appropriate statistics for all operations considered in a

[2] In the actual implementation of the framework in Python, operations may return None or raise an exception to indicate failure. Such implementations can be wrapped to comply with the requirements of our framework and we can adjust the choice of what value is used to encode $\perp$ to simplify adoption of our framework.

search and from then on only operate by testing preconditions over the profiles, thereby reducing the cost of individual applicability tests from $O(|C|)$ to $O(|P_C|)$ where P_C is the profile for the column whose size is typically independent of $|C|$.

Normalized preconditions. In this section, we only consider preconditions in the normal form shown below where p and $\vec{p} = p_1, \ldots, p_m$ are parameters of the operation for which we are constructing a precondition, k is a constant, and f is a black-box function:

$$f(x, \vec{p}) \geq k \qquad f(x, \vec{p}) \geq p \qquad \textbf{(precondition normal form)}$$

While this may seem limiting at first, note that because we do not place any restrictions on f, any precondition we have observed in practice can be translated trivially into this normal form. For example, if ψ is an expression returning a Boolean value, i.e., $\psi := e(x, \vec{p})$, then we can normalize it using the indicator function $\mathbb{1}[\cdot]$: $\psi_{norm} := f'(x, \vec{p}) \geq 1$ for $f' := \mathbb{1}[e(x, \vec{p})]$; if $\psi := f(x, \vec{p}) \leq k$ we can normalize it using $f'(x, \vec{p}) := -f(x, \vec{p})$, and so on.

Column profiles. Given a normalized precondition ψ with function f and a threshold θ for the fraction of column values that should fulfill ψ, we can determine whether $\Psi_\theta(X, \vec{p})$ holds based on the value of the $1-\theta$-tile of $f(v, \vec{p})$ over column C. That is, given a set $\mathbb{P}$ of possible settings for $\vec{p}$ we have to compute the following statistic for each $\vec{c} \in \mathbb{P}$:

$$P_C[f][\vec{c}] := \textsc{qtile}_{1-\theta}\left(\{f(C[i], \vec{c}) \mid i \in [1, |C|]\}\right) \tag{3}$$

Here, $\textsc{qtile}_y(S)$ for $y \in [0, 1]$ computes the y-tile of ordered set S, i.e., the smallest value such that precisely a fraction $\lfloor y \cdot |S| \rfloor$ of values in S are smaller than or equal to $\textsc{qtile}_y(S)$. Given a set of normalized preconditions $\{\psi_i\}_{i=1}^{l}$, a *column profile* P_C for a column C then collects all relevant statistics needed to evaluate the preconditions in $\{\psi_i\}_{i=1}^{l}$. That is, P_C stores parameterized statistics for every function f appearing in some ψ_i and every $\vec{c} \in \mathbb{P}$ for each ψ_i.

Profile-based precondition. Given a normalized precondition $\psi := f(x, \vec{p}) \geq k$, we can derive a *profile-based precondition* $\hat{\Psi}$ that only accesses the profile of a column as follows:

$$\hat{\Psi}_\theta(X, \vec{p}) := P_X[f][\vec{p}] \geq k \qquad \textbf{(profile-based precondition)}$$

Critically, $\hat{\Psi}$ is equivalent to Ψ.

Proposition 1 (Profile Precondition Correctness). *Given a precondition $\psi := f(x, \vec{p}) \geq k$ in normal form, we have:*

$$\hat{\Psi}_\theta(X, \vec{p}) \equiv \Psi_\theta(X, \vec{p})$$

As an example of the generation of profile preconditions consider the precondition ψ' from Equation (1) for operation $\text{substr}(x, b, e)$ with a threshold of $\theta = 0.5$. Furthermore, consider the column shown below:

$C = \langle$`'Smith, Sr.'`, `'Bob,S'`, `'Johnson'`, `'Jobs, PhD'`, `'Bale'`$\rangle$

Precondition $\Psi'_{0.5}(C, 1, 6)$ evaluates to true as 3 out of 5 values (more than 50%) fulfill the condition as these values have more than $e = 6$ characters. This condition is almost in normal form, we can simply substitute $\text{len}(x)$ with $\text{len}(x) - 1$ to get:

$$\psi'''(x, b, e) := \text{len}(x) - 1 \geq e$$

Note that none of the parameters passed to substr are used in the LHS of the comparison. Let $f = \text{len}(x) - 1$, applying f to all values in a column values and taking the quantile, we get the profile statistic for ψ''' as a single value:

$$P_C[\text{len}] = \textsc{qtile}_{0.5}(\{9, 4, 6, 8, 3\}) = 6.$$

As claimed in Prop. 1, we get $\hat{\Psi}'''_{0.5}(C, 1, 6) = P_C[\text{len}] \geq 6 = \textbf{\textit{true}}$.

Now consider precondition ψ'' from Equation (2) for operation $\text{split}(x, d, j)$ which is already in normal form and requires that the input string has to contain at least j instances of delimiter d: $\psi'' := \text{count}(x, d) \geq j$. Note that parameter d is passed to count. Thus, the profile for the column will contain the 0.5-tile for all values of d considered by the search algorithm. For example, if we assume that `' , '` and `' ; '` are considered then we get

$$P_C[\text{count}][\text{`' , '`}] = 3 \qquad P_C[\text{count}][\text{`' ; '`}] = 0,$$

as 3 values contain the character `' , '`, but none contain `' ; '`. Thus, the (profile-based) precondition fails for `' ; '`, but succeeds for `' , '`.

Note that storing quantiles for every possible parameter setting in $\mathbb{P}$ for a precondition is feasible, because typically $\mathbb{P}$ for arguments to f is small. In instances where this is not the case, we could group parameter settings and maintain a lower bound on f across all settings in the group as a smaller profile statistic with potentially lower pruning power. We leave further exploration of this idea to future work. Furthermore, profiles are specific to thresholds θ and we allow the thresholds to vary across preconditions. While in most settings such thresholds would be set globally, if we want to allow search algorithms to adjust thresholds during the search without requiring our framework to recompute column profiles whenever a threshold is adjusted, we could precompute profiles for a small number of thresholds and then use the next larger precomputed threshold instead of the threshold provided to us by the search algorithm. For example, if we only compute profiles for $\theta \in \{0.1, 0.2, \ldots, 0.9, 1.0\}$ then for threshold 0.35 we would instead use the profile computed for $\theta = 0.4$ as it results in a more restrictive condition.

2.3 Integration with Transformation Search

We model transformation workflows $\mathcal{T}$ as directed acyclic multigraphs (DAGs) whose nodes are atomic operations and where edges labeled (i, j) denote dataflow: the i^{th} output of an operation is passed as the j^{th} input of the next operation (here we consider the more general definition of operations that can have multiple input and output columns). Our framework integrates with an existing search algorithm in two ways, both requiring operation preconditions as part of the input.

Candidate-level pruning. The search algorithm generates candidate operations for a given column C and parameter assignment $\vec{c}$. Before executing a candidate operation $\text{op}(C, \vec{c})$, it calls $\textsc{isapplicable}(\text{op}, C, \vec{c})$ from our framework to test whether the candidate should be pruned. In turn isapplicable generates a column profile unless one exists already and then uses the profile-based preconditions to determine whether the operation is guaranteed to fail. When using isapplicable in this fashion, our framework reduces the runtime of evaluating candidate operations as profiles have to be created only once upfront and preconditions have to only be checked on profiles instead of full columns.

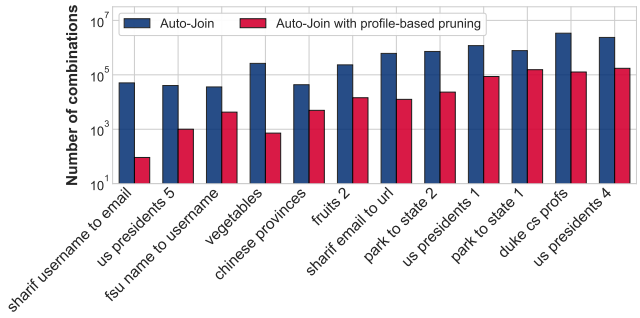

Figure 2: Size of the explored search space (number of instantiated operators) for Vanilla Auto-Join versus Auto-Join with our profile-based pruning techniques.

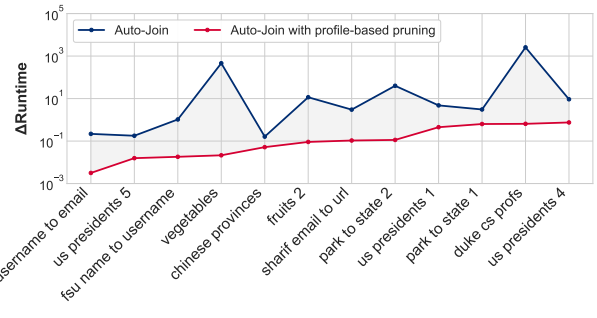

Figure 3: Runtime of Vanilla Auto-Join versus Auto-Join using our framework on the benchmark tasks from [16]. Our approach improves performance by one to over four OOM.

Parameter-space pruning. The search algorithm can also use PRUNEPARAMETERS which returns a subset of an input parameter space $\mathbb{P}$ for an operation. In the worst case, this requires testing all settings in $\mathbb{P}$. However, for preconditions that are monotone or antimonotone in a parameter p, we can prune the search space for p using binary search. For example, $\psi'(x, \mathrm{b}, \mathrm{e}) \coloneqq \mathsf{len}(x) > \mathrm{e}$ from Equation (1) is monotone in parameter e. If the maximum string length in a column is 5, then all substr candidates with end position e greater than 5 can be pruned without evaluating the profile precondition explicitly for each such candidate.

3 CASE STUDY: AUTO-JOIN

As a proof of feasibility, we apply our framework to a joinability task using Auto-Join [16] as the transformation search algorithm.

Auto-Join's search algorithm. Auto-Join takes a source table T_s and a target table T_t, and searches for a transformation that makes their key columns equi-joinable. It identifies candidate joinable row pairs via q-gram matches and then uses them as input-output examples to find a transformation. The search algorithm enumerates candidate operations and their parameter instantiations, returning the transformation with the minimum number of operations that is consistent with the examples.

Operations & preconditions. Auto-Join supports four operations: Constant, Substr, SplitSubstr, SplitSplitSubstr. We manually constructed preconditions for these operations, some of which were already discussed as examples in the previous section. For each source column, we compute profiles with a pruning threshold $\theta = 1$.

Integration. We integrate our framework into Auto-Join by adding a call to our library's PRUNEPARAMETERS primitive after Auto-Join enumerates the candidate parameter space for an operation. We specify preconditions as Python lambdas alongside each operation's definition, which required fewer than 10 lines of code.

Setup. We ran experiments on an Apple MacBook Pro M3, with 8 GB RAM, running macOS 15.5. Our framework is implemented in Python 3.14. We used the Web benchmark from [16], consisting of 31 transformation tasks with ground truth, with a one-hour-per-task timeout, averaged over three runs (tasks that timed out are excluded).

Results. We compare our approach with vanilla Auto-Join on two metrics: runtime and search-space reduction. Profile-based pruning improves runtime on every task (Figure 3), with an average speedup of $\sim 2000\times$ and a maximum of $\sim 22,000\times$ on *vegetables*, where vanilla Auto-Join takes 458 seconds and our approach takes 0.02 seconds. Even on tasks where Auto-Join is already fast (0.2s on `sharif username to email`), pruning still delivers a $78\times$ speedup. These gains come from skipping transformations whose preconditions are bound to fail based on the column profile alone, avoiding both candidate instantiation and the column scans needed to evaluate them. We expect the relative savings to increase further at data-lake scale, where the cost of evaluating infeasible candidates over full columns dominates. Figure 2 shows that our approach reduces the number of instantiated operator-parameter combinations across all benchmark tasks. In addition to search space reduction, runtime improvements are also affected by the cost of evaluating operations on a column, e.g., dataset `duke cs profs` has long strings, so runtime improvements far exceed the reduction in search space size while at the other end of the spectrum for `sharif username to email` the cost of evaluating string operator is low and runtime savings underperform the reduction in search space size.

4 CONCLUSION & FUTURE WORK

Automated search for transformations is essential for many applications including programming-by-example in spreadsheets, schema matching, data discovery, and many others. However, developing general and efficient transformation search techniques has been challenging due to the large search space and diversity of operations used in transformations. In this work, we present a framework that, based on operator preconditions, automatically selects lightweight statistics to maintain as *column profiles* and generates conditions on profiles for pruning candidate operations. Our techniques can be easily integrated into any transformation search algorithm to improve its performance. In future work, we will evaluate our approach on a wider selection of transformation search algorithms, consider probabilistic pruning based on column profiles computed on samples for very large columns, and extend our approach to more general classes of transformations, such as constraint-based cleaning operations. We will also explore our framework's integration with LLM-based transformation systems, to formally verify transformations they produce and improve their scalability [11].

REFERENCES

[1] Ziawasch Abedjan, John Morcos, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, and Michael Stonebraker. 2016. Dataxformer: A robust transformation discovery system. In *ICDE*. 1134–1145.

[2] Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: a powerful approach to weakest preconditions. *SIGPLAN Not.* 44, 6 (2009), 363–374.

[3] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: dynamic symbolic execution for invariant inference. In *ICSE*. 281–290.

[4] Elizabeth Dinella, Shuvendu K. Lahiri, and Mayur Naik. 2024. Inferring Natural Preconditions via Program Transformation. In *FSE*. 657–658.

[5] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *POPL*. 317–330.

[6] Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek Narasayya, and Surajit Chaudhuri. 2018. Transform-data-by-example (TDE) an extensible search engine for data transformations. *PVLDB* 11, 10 (2018), 1165–1177.

[7] Zhongjun Jin, Michael R Anderson, Michael Cafarella, and HV Jagadish. 2017. Foofah: Transforming data by example. In *SIGMOD*. 683–698.

[8] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive visual specification of data transformation scripts. In *CHI*. 3363–3372.

[9] Daragh King, Vasileios Koutavas, and Laura Kovács. 2026. LLMs and fuzzing in tandem: a new approach to automatically generating weakest preconditions. *STTT* 28, 3 (2026), 317–328.

[10] Eugenie Lai, Yeye He, and Surajit Chaudhuri. 2025. Auto-Prep: Holistic Prediction of Data Preparation Steps for Self-Service Business Intelligence. *PVLDB* 18, 7 (2025), 2212–2225.

[11] Changlun Li, Chenyu Yang, Yuyu Luo, Ju Fan, and Nan Tang. 2025. Weak-to-strong prompts with lightweight-to-powerful llms for high-accuracy, low-cost, and explainable data transformation. *PVLDB* 18, 8 (2025), 2371–2384.

[12] Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd D. Millstein. 2018. FlashProfile: A Framework for Synthesizing Data Profiles. In *OOPSLA*. 1 – 28.

[13] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *OOPSLA*. 107–126.

[14] Rishabh Singh. 2016. BlinkFill: Semi-supervised Programming by Example for Syntactic String Transformations. In *PVLDB*, Vol. 9. 816–827.

[15] Fan Yang, Xuqian Ma, Shuling Wang, Xiong Xu, Qinxiang Cao, Naijun Zhan, Xiaofeng Li, and Bin Gu. 2025. Integrating Symbolic Execution with LLMs for Automated Generation of Program Specifications. *arXiv preprint arXiv:2506.09550* (2025).

[16] Erkang Zhu, Yeye He, and Surajit Chaudhuri. 2017. Auto-Join: Joining Tables by Leveraging Transformations. In *PVLDB*, Vol. 10. 1034–1045.

[17] Aslihan Özmen, Mahdi Esmailoghli, and Ziawasch Abedjan. 2021. Combining Programming-by-Example with Transformation Discovery from large Databases. In *BTW*. 313–324.