Auto-Join: Joining Tables by Leveraging Transformations

Erkang Zhu*
University of Toronto
ekzhu@cs.toronto.edu

Yeye He
Microsoft Research
yeyehe@microsoft.com

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

ABSTRACT

Traditional equi-join relies solely on string equality comparisons to perform joins. However, in scenarios such as ad-hoc data analysis in spreadsheets, users increasingly need to join tables whose join-columns are from the same semantic domain but use different textual representations, for which transformations are needed before equi-join can be performed. We developed Auto-Join, a system that can automatically search over a rich space of operators to compose a transformation program, whose execution makes input tables equi-join-able. We developed an optimal sampling strategy that allows Auto-Join to scale to large datasets efficiently, while ensuring joins succeed with high probability. Our evaluation using real test cases collected from both public web tables and proprietary enterprise tables shows that the proposed system performs the desired transformation joins efficiently and with high quality.

1. INTRODUCTION

Join performs the powerful operation of combining records from two or more tables together, and is of key importance to relational databases. It is extensively used in modern relational database systems, as well as data analysis software such as Power Query for Excel [5], Informatica [3], etc.

Most existing commercial systems only support *equi-join* that uses string equality comparisons. While equi-join works well in curated settings such as data warehousing, where data are extensively cleansed and prepared through ETL, it often falls short in scenarios where data is less curated. For example, analysts today often perform *one-off*, *ad-hoc* data analysis, where they need to correlate/join datasets from different sources whose key columns are formatted differently. Requiring support staffs to perform ETL for such ad-hoc scenarios is often too slow and expensive – automating these joins for end-users has become increasingly important.

Figure 1 shows such an example. An analyst has a table on the left in her spreadsheets about US presidents and popular votes they won in elections. She uses table search engines (such as Google Web Tables [2] or Microsoft Power Query [4]) to find an additional table on the right, that has information about their job approval rating. Now she wants to join these two tables so that she can correlate them. However, the `name` columns of the two tables use different representations – the one on the left uses first-name followed by last-name, while the one on the right uses last-name,

President	Popular Vote
Barack Obama	52.93%
George W. Bush	47.87%
Bill Clinton	43.01%
George H. W. Bush	53.37%
Ronald Reagan	50.75%

President	Approval Rating
Obama, Barack(1961-)	47.0
Bush, George W.(1946-)	49.4
Clinton, Bill(1946-)	55.1
Bush, George H. W.(1924-)	60.9
Reagan, Ronald(1911- 2004)	52.8

Figure 1: (left): US presidents and popular votes. (right): US presidents and job approval rating. The right table uses last-name, comma, first-name, with (year-of-birth and year-of-death).

Name	Title
Suhela Chowdhury	Principal
Maureen Paluzzi	Instructor
Missy Payne	Instructor
Carolyn Craddock	Admin
Kelly Moore	Instructor

Email	School
schowdhury@forsyth.k12.ga.us	Big Creek
mpaluzzi@forsyth.k12.ga.us	Brookwood
mipayne@forsyth.k12.ga.us	Chattahoo
ccraddock@forsyth.k12.ga.us	Chestatee
kmoore@forsyth.k12.ga.us	Princeville

Figure 2: (left): Name and job titles in school. (right): Email and school districts. `Email` from the right table can be generated from `name` in the left by concatenating first-initials, last-names, and '@forsynth.k12.ga.us'.

ATU	Manager Alias
France.01	V-JOHH
France.03	JOFORD
United States.01	RICHT
United States.02	MICHM
United States.03	ANDYW

Sub-ATU	Segment
France.01.MIX	SMB
United States.01.Government	Major
United States.01.Education	AM EPG
United States.03.PS-LRG	TM SMS&P
United States.04.Retail	AM SMS&P

Figure 3: (left): `ATU` name (for area team unit). (right): `Sub-ATU` names organized under `ATU`.

ID	Session Name
UBAX01	AXUG General Session
UBAX02	How2 Session
UBAX03	Master Planning Session
UBAX04	Financial Reporting
UBAX05	Master Planning Session

Full Session Name	Month
[UBAX01] AXUG General Session	Mar
[UBAX02] How2 Session	Apr
[UBAX03] Master Planning Session	Apr
[UBAX04] Financial Reporting	Oct
[UBAX05] Master Planning Session	Dec

Figure 4: (left): `ID` and `session name` in separate fields. (right): Concatenated `full session name`.

*Work done at Microsoft Research

ACM ISBN .
DOI:

comma, first-name, with additional year-of-birth information in parenthesis. Today's popular data analysis software such as Power Query for Excel [5] and Informatica [3] only support equi-join and would fail to work on these two tables. The analyst would have to either write her own transformation program so that the name representations become consistent for equi-join[1], or ask technology support staffs to perform such a transformation.

Figure 2 shows another case with two real tables collected from the web, where the table on the left uses teachers' full names, while the table on the right only has email addresses as the key column. Note that in this case because email aliases can be generated by taking first-initials and concatenating with last names, there exists a clear join relationship between the two tables. Equi-join, however, would again fail to work in this case.

Scenarios like these are increasingly common in ad-hoc data analysis, especially when analysts need to bring in data from different sources, such as public datasets discovered from table search engines.

It is worth noting that this join problem exists not only for public data sets like web tables, but also in enterprise data tables such as Excel files. Figure 3 shows a pair of real spreadsheet tables from a corpus of Excel files crawled in a large IT company. The `ATU` (area-team-unit) column on the left can be joined with `Sub-ATU` on the right in a hierarchical manner. The join can be produced by simply taking the first two components of `Sub-ATU`, which can then equi-join with `ATU` in a hierarchical N:1 manner. Figure 4 shows another example from enterprise spreadsheets. An enterprise worker who wants to combine these two tables cannot use equi-join. However, if we concatenate `id` and `session name` in the left table with appropriate brackets, the two tables can then be equi-joined.

Joining tables with different formats and representations is a ubiquitous problem. Notice that in these cases, simple syntactic transformations (e.g., substring, split, concatenation) can often be applied to make equi-join possible. While humans looking at these tables can intuitively identify transformations needed to join, existing commercial systems that use equi-joins will fail. In this work, our goal is to automate the discovery of syntactic transformations needed such that two tables with different representations can be joined with the click of a button. Note that because such transformations are driven by end-user tools such as spreadsheets, a significant technical challenge is to make such transformation-based join very efficient and at interactive speed.

Existing solutions: A few existing approaches may be used for this problem, but none produces satisfactory results.

Program transformations manually. A straightforward approach is to ask users to manually write transformation programs that produce derived columns, with which tables can be equi-joined. This is clearly inconvenient, and potentially too difficult for end-users in spreadsheets such as data analysts or data scientists. We want to automate this process without asking users to write programs.

Fuzzy join. Since rows that join under syntactic transformations typically have substantial substring overlaps, an alternative approach is to use fuzzy join [8]. The challenge is that fuzzy join has a wide range of parameters (tokenization, distance-function, thresholds, etc.) that need to be configured appropriately to work well. The ideal configuration can vary significantly from case to case, and is difficult for users to determine. For instance, for Figure 1, one should tokenize by words, but that tokenization will fail completely for Figure 2, which requires q-gram tokenization.

Furthermore, even when fuzzy join is configured perfectly, it may still produce incorrect results due to its fuzzy and imprecise nature. For example, in Figure 1, if we tokenize by words, and want to join `Ronald Reagan` and `Reagan, Ronald(1911-2004)`, the threshold for Jaccard distance should be at least 0.66 (this pair has a distance of $1.0 - \frac{1}{3} = 0.66$). However, using a threshold of 0.66 will also join `George W. Bush` with `Bush, George H. W.(1924-)` (where the distance is $1.0 - \frac{3}{5} = 0.4$), which is incorrect. The root cause here is that fuzzy join uses an imprecise representation and simple threshold-based decision-boundary that is difficult to be always correct. In comparison, there are many cases where regularity of structures in data values exists (e.g. Figure 1-4), and for those cases using consistent transformations for equi-join complements fuzzy-join by overcoming its shortcomings mentioned above.

Substring Matching [22]. Warren and Tompa [22] proposed a technique to translate schema between database tables, which is applicable to joins and is the only published technique that we are aware of that can produce transformation-based joins given two tables. However, the types of transformations they considered are rather limited (e.g., no string-split and component based indexing), and as a result their approach is not expressive enough to handle many real join tasks we encountered. As we will show in our experiments, their approach can handle less than 30% of the join cases we collected.

Our contributions. We make the following contributions in the Auto-Join system.

- We propose to automate transformation-based joins using a novel Auto-Join algorithm. Our technique leverages substring indexes to efficiently identify promising row pairs that can potentially join. We then use these row pairs to automatically learn *minimum-complexity* programs whose execution can lead to equi-joins.
- In order to scale Auto-Join to large tables while still maintaining interactive speed, we design a sampling scheme that minimizes the number of rows sampled while guaranteeing success of transformation-join with high probability.
- We are the first to compile benchmarks with over 70 cases requiring transformation joins. We label each test case with a fine-grained row-level ground truth. Our results show that the proposed approach produces joins with higher quality than existing approaches.

2. PROBLEM OVERVIEW

Our objective is to automate transformation-joins by generating the transformations that are needed for equi-joins. Specifically, we want to transform columns of one table through a sequence of string-based syntactic operations, such that the derived columns can be equi-joined with another table. Example 1 gives such an example.

Example 1. In Figure 1, there exists a transformation whose application on the right table can lead to equi-joining with the left table. We illustrate the sequence of opera-

[1]Such behavior is observed in logs collected from a commercial data preparation software – for certain datasets users perform sequences of transformations in order to enable equi-join.

tions in the transformation using the first row {[Obama, Barack(1961-)], [47.0]} as an example.

1. Input row X with two elements: {[Obama, Barack(1961-)], [47.0]}
2. Take the first item $X[0]$, Split by "(", produce Y: {[Obama, Barack], [1961-)]}
3. Take the first item $Y[0]$, Split by ",", produce Z: {[Obama], [Barack]}
4. Takes Substring [1:] from $Z[1]$, produce T: [Barack]
5. Concat T, a constant string " " and $Z[0]$, produce [Barack Obama]

This derived value can then be equi-joined with the first row in the left table in Figure 1. It can be verified that the same transformation can also be applied on other rows in the right table to equi-join with the President column of the left table.

As discussed earlier, while such transformations can be written manually as a program (e.g., a Python script), doing so is cumbersome and requires programming skills. We would like to automate the generation of the transformations required for equi-joins. We call this the *transformation join problem*.

Definition 1. *Transformation Join Problem*: Given two tables T_s, T_t, and a predefined set of operators Ω, find a transformation $P = o_1 \cdot o_2 \cdot o_3 \cdot \ldots o_n$, using operators $o_i \in \Omega$, such that $P(T_s)$ can equi-join with key columns of T_t.

Here each transformation P is composed of a sequence of operators in Ω, where the output of one operator is the input of the next. For the purpose of transformation-join, we have identified a small set of operators that are sufficient for almost all join scenarios we encountered.

$$\Omega = \{\text{Split, Concat, Substring, Constant, SelectK}\} \quad (1)$$

This set of operators Ω can be expanded to handle additional requirements as needed.

In this definition, because we require $P(T_s)$ to equi-join with key columns of T_t, the types of join we consider are implicitly constrained to be 1:1 join (key:key) or N:1 join (foreign-key:key). This property is important because it ensures that the joins we automatically generate are likely to be useful; relaxing this constraint often leads to N:M joins that are false positives (e.g., join by taking three random characters from two columns of values).

Also observe that we apply transformations on one table T_s in order to equi-join with another table T_t. We refer to the table T_s as the *source table*, and T_t as the *target table*, respectively.

Because in our problem, we are only given two tables with no prior knowledge of which table is the source and which is the target, we try to generate transformations in both directions. In Example 1 for instance, the right table is used as the source table. Changing direction in this case does not generate a transformation-join because year-of-birth is not present in the left table. Advanced handling of composite columns will be discussed in Section 3.3.

Solution Overview. Our Auto-Join system has three high-level steps.

Step 1: Find Joinable Row Pairs. In our problem, we only take two tables as input, without knowing which row from T_s should join with which row from T_t. Generating transformations without such knowledge would be exceedingly slow, due to the quadratic combinations of rows in two tables that can potentially join.

So in the first stage, we attempt to "guess" the pairs of rows from the two tables that can potentially join. We leverage the observation that unique q-grams are indicative of possible join relationships (e.g. Obama), and develop an efficient search algorithm for joinable row pairs.

Step 2: Learn Transformation. Once we obtain enough row pairs that can potentially join, we learn a transformation that uses rows from T_s as input and generates output that can equi-join with key columns of T_t. In Example 1 for instance, the desired transformation uses {[Obama, Barack (1961-)], [47.0] } as input, and produces [Barack Obama] as output to join with the key column of the left table. Since there likely exists many possible transformations for one particular input/output row pair, we use multiple examples to reduce the space of feasible transformations, and then pick the one with minimum complexity that likely best generalizes the observed examples. This learning process is repeated many times using different combinations of row pairs, and the transformation that joins the largest fraction of rows in T_t is produced as the result.

Step 3: Constrained Fuzzy Join. In certain cases such as tables on the web, the input tables may have inconsistent value representations or dirty values. For example, in Figure 2, the second row of the right table uses mi-payne@forsyth.k12.ga.us, instead of first-initial concatenated by last-name like other rows (which would produce mpayne@forsyth.k12.ga.us). Thus, transformations may miss this joinable row pair. As an additional step to improve recall, we develop a mechanism that automatically finds a fuzzy-join with optimized configuration to maximize additional rows to join, without breaking the join cardinality constraints (i.e., 1:1 or N:1). This improves the coverage of joins on dirty tables, and is a technique of independent interest for the important problem of automatically optimizing fuzzy join.

Symbol	Description
T_s, T_t	T_s is the source table, T_t is the target table.
R_s, R_t	A row in T_s and T_t, respectively
C_s, C_t	A column in T_s and T_t, respectively
$\mathcal{Q}_q(v)$	The q-grams of a string value v
$\mathcal{Q}_q(C_s)$	The multi-set of all q-grams in C_s
$\mathcal{Q}_q(C_t)$	The multi-set of all q-grams in C_t

Table 1: Notations used for analyzing q-gram matches

3. AUTO-JOIN BY TRANSFORMATIONS

In this section we discuss the first two steps of Auto-Join: (1) finding joinable row pairs, and (2) learn transformations that generalize the examples observed in these row pairs.

3.1 Finding Joinable Row Pairs

Let P be the desired transformation, such that $P(T_s)$ can equi-join with T_t as in Definition 1. Pairs of rows that join

under this transformation P is termed as *joinable row pair*. For instance, the first row from the left and right table in Figure 1 is a joinable row pair.

Since users do not provide joinable row pair as input to our system (which is cumbersome to provide especially in large tables), in this section we explain our approach for "guessing" joinable row pairs candidates from the two table through unique q-grams matches. Note that finding such row pairs is important, because trying the quadratic number of row combinations exhaustively is too computationally expensive to be interactive.

We leverage the observation that the set of operations Ω considered for transformation-join (Equation 1) tend to preserve local *q-gram*. A q-gram [7] of a string v is a substring of v with q consecutive characters. A complete q-gram tokenization of v, denoted as $\mathcal{Q}_q(v)$, is the set of all possible q-grams of v. For example:

$$\mathcal{Q}_5(\texttt{Database}) = \{\texttt{Datab}, \texttt{ataba}, \texttt{tabas}, \texttt{abase}\}$$

q-grams have been widely used for string similarity and language modeling, among other applications.

Operations required for transformation-joins in Ω all tend to preserve sequences of local q-grams, which is the key property we exploit to find joinable row pairs.

3.1.1 1-to-1 q-Gram Match

Intuitively, if we can find a *unique* q-gram that only occurs once in T_s and T_t, then this pair of rows is very likely to be a joinable row pair (e.g., q-gram `Barack` in Figure 1). We start by discussing such 1-to-1 matches, and show why they are likely joinable row pairs using a probabilistic argument.

It is known that q-grams in texts generally follows power-law model [6, 14]. We conducted a similar experiment on a large table corpus with over 100M web tables and observed similar power law results B. For such power law distributions, the probability mass function for a q-gram whose frequency ranks at position k among all q-grams, denoted by $p_q(k)$, is typically modeled as [6, 14]

$$p_q(k) = \frac{\frac{1}{k^{s_q}}}{\sum_{z=1}^{N} \frac{1}{z^{s_q}}} \tag{2}$$

Here k is the rank of a q-gram by frequency, N is the total number of q-grams, and s_q is a constant for a given q. These power-law distributions empirically fit well with real data [6].

Given a pair of tables whose q-grams are randomly drawn from such a power-law distribution, we can show that it is extremely unlikely that a q-gram appears exactly once in both tables *by chance* (for reasonably large tables, e.g., $N > 100$).

PROPOSITION 1. *Given two columns C_s and C_t from tables T_s and T_t respectively, each with N q-grams from an alphabet of size $|\Sigma|$ that follow the power-law distribution above. The probability that a q-gram appears exactly once in both C_s and C_t by chance is bounded from above by the following.*

$$\sum_{k=1}^{|\Sigma|^q} \Big((1 - p_q(k))^{N-1} \cdot p_q(k)\Big)^2 \tag{3}$$

A proof this result can be found in Appendix C.

For $q = 6$, $N = 100$, $|\Sigma| = 52$, and using the s_q defined in [6], the probability of any 6-gram appearing exactly once by chance on both columns is very small (< 0.00017). This quantity will in fact grow exponentially small for larger N (typical tables have at least thousands q-grams).

Given this result, we can conclude that if we do encounter unique 1-to-1 q-gram matches from two tables, they are unlikely coincidence but the result of certain relationships.

Let $\mathcal{Q}_q(C)$ be the multi-set of all the q-grams of distinct values [2] in column C; and let $F_q(g, C)$ be the number of occurrences of a q-gram $g \in \mathcal{Q}_q(C)$. Let v_s and v_t be the cell value at row R_s column C_s in T_s and row R_t column C_t in T_t, respectively. We define *1-to-1 q-gram matches* as follows.

DEFINITION 2. Let g be a q-gram with $g \in \mathcal{Q}_q(v_s)$ and $g \in \mathcal{Q}_q(v_t)$. If $F_q(g, C_s) = 1$ and $F_q(g, C_t) = 1$, then g is a *1-to-1 q-gram match* between row pair R_s and R_t with respect to the pair of column C_s and C_t.

As we have discussed, matches that are 1-to-1 q-gram are likely joinable row pairs.

EXAMPLE 2. Given two tables in Figure 1, the 6-gram `Barack` appears only once in both tables, and the corresponding rows in these two tables are indeed a joinable row pair. The same is true for q-grams like `chowdury` in Figure 2, `France.01` in Figure 3 and `UBAX01` in Figure 4, etc.

As the reader will see in the experimental results (Section 6.3), using 1-to-1 q-gram matches as joining row pairs leads to a precision of 95.6% in a real-world benchmark.

3.1.2 General n-to-m q-Gram Match

q-gram matches that are 1-to-1 are desirable special cases. In general we have *n-to-m q-gram matches*.

DEFINITION 3. Let g be a q-gram with $F(g, C_s) = n \geq 1$ and $F(g, C_t) = m \geq 1$, then g is a *n-to-m q-gram match* for corresponding rows with respect to the pair of column C_s and C_t.

Compared to 1-to-1 q-gram matches that almost always identify a joinable row pair, the probability that a row pair identified by n-to-m matches is a true joinable row pair is roughly $\frac{1}{n \cdot m}$. We use $\frac{1}{n \cdot m}$ to quantify the "goodness" of the match.

Note that the ideal q to identify n-to-m matches with small n and m can vary significantly in different cases.

EXAMPLE 3. For the tables in Figure 1, if we use 6-grams for value `Barack Obama`, we get an ideal 1-to-1 match of `Barack` between the first rows of these two tables. However, if we also use 6-gram for the second row `George W. Bush`, then the best we could generate is a 2-to-2 match using the 6-gram `George`, between the second and fourth rows of these two tables, respectively.

For `George W. Bush`, the ideal q should be 9, since the 9-gram `George W.` could produce a 1-to-1 match. However, if we use 9-grams for the first row `Barack Obama`, we would fail to generate any q-gram match.

The ideal q is not known a priori and needs to be searched.

3.1.3 Efficient Search of q-Gram Matches

A simple algorithm for finding ideal q-gram matches (with small n and m) would conceptually operate as follows: (1) for every cell from one table, (2) for all possible settings of q, (3) for each q-gram in the resulting tokenization, (4)

[2] We remove possible duplicates in T_s columns since they are potentially foreign keys.

iterate through all values in the this table to find the number of q-gram matches, denoted as n; (5) iterate through all values in the other table to find the number of q-gram matches, denoted as m. The resulting match can be declared as an n-to-m match. This is clearly inefficient and would fail to make the desired join interactive. In this section we describe efficient techniques we use to search for unique q-gram matches.

First, we build a *suffix array index* [19] for every column in the source table and each column of the target table, so that instead of using step (4) and (5) above, we can search with logarithmic complexity. A suffix array index is built by creating a sorted array of the suffixes of all values in a column. Given a query q-gram, matches can be found by using binary search over the sorted array and looking for prefixes of the suffixes that match the query exactly. The complexity of probing a q-gram in the index is $O(\log S)$, where S is the number of unique suffixes. We probe each q-gram once in both tables, to find the number of matches n and m. An example of using suffix array index can be found in Appendix A.

Using suffix array significantly improves search efficiency for a given q-gram. However, for a cell value v, we still need to test all possible q-grams. To efficiently find the best q-gram (with the highest $\frac{1}{nm}$ score), we express the optimal q-gram g^* as the best prefix of all possible suffixes of v.

$$g^* = \underset{\forall g \in \textsc{Prefixes}(u), u \in \textsc{Suffixes}(v)}{\arg\max} \frac{1}{nm} \tag{4}$$

Where $n = F(g, C_s) > 0$ and $m = F(g, C_t) > 0$ are the number of matches in column C_s and C_t, respectively. We leverage a monotonicity property described below.

Proposition 2. *Let g_u^q be a prefix of a suffix u with length q. As the length increases by 1 and g_u^q extends at the end, the $\frac{1}{nm}$ score of the longer prefix g_u^{q+1} is monotonically non-increasing, or $F(g_u^{q+1}, C_s) \leq F(g_u^q, C_s)$ and $F(g_u^{q+1}, C_t) \leq F(g_u^q, C_t)$.*

A proof of this can be found in Appendix C.

Given Proposition 2, for every suffix u we can find $g_u^{q^*}$ by looking for the longest prefix with matches in C_t using binary search. The global optimal g^* can be found by taking the $g_u^{q^*}$ with the highest score for all u.

Example 4. In Figure 1, for the value `George W. Bush`, we iterate through all its suffixes (e.g., "`George W. Bush`", "`eorge W. Bush`", etc.). For each suffix, we test their prefixes using binary search to find the one with the best score (the longest prefix with match), from which we select the best prefix. In this case the prefix "`George W.`" for the first suffix is the best g^*.

With this 1-to-1 match, we can determine that the first rows from the left/right tables in Figure 1 are joinable row pairs. Similarly the second rows from the two tables are also joinable row pairs, etc.

Because of the use of suffix array indexes and binary search, our overall search complexity is $O(|v| \log |v| \log S)$, which is orders of magnitude more efficient than the simple method discussed at the beginning of this section.

3.1.4 Putting it together: Find Joinable Rows

Algorithm 1 gives the high-level pseudo code for this step of finding joinable row pairs. For each pair of C_s and C_t, we iterate through distinct value $v \in C_s$, and use OptimalQGram (the procedure discussed above) to efficiently find the best q-gram match and its associated row pairs. Finally, row pairs connected together by q-gram matches are grouped by C_s and C_t, and ordered by their match scores. Details of this step can be found in Appendix E.1.

Algorithm 1 Find joinable row pairs.

```
1: function FindJoinableRowPairs(T_s, T_t)
2:     M ← {}                                          ▷ q-gram matches
3:     for all C_s ∈ T_s do
4:         for all C_t ∈ KeyColumns(T_t) do
5:             for all v ∈ C_s do
6:                 {g*, score, R_s, R_t} ← OptimalQGram(v, C_t)
7:                 M ← ∪{(g*, score, R_s, R_t, C_s, C_t)}
8:     return M, GroupBy(C_s, C_t), OrderBy(score)
```

It is worth noting that we group matches by the column pairs from which the matches are produced. This is because we want to produce consistent transformations on certain columns C_s in T_s, so that the results can equi-join with columns C_t in T_t. As such, matches found in different column pairs are good indications that they belong to different transformation-join relationships, as illustrated by the following example.

Example 5. In Figure 2, in addition to q-gram matches between the columns `Name` and `Email`, there is a q-gram match for `Princ`, which matches `Principal` in the first row in the `Title` column from the left table, and `Princeville` in the last row in the `School` column from the right table. However, matching row pairs produced between `Title` and `School` can be used to produce a transformation-join relationship between these columns (if one exists), which should be treated separately from one produced using matches between the `Name` and `Email` columns.

3.2 Transformation Learning

Given joinable row pairs $\{(R_s, R_t)\}$ produced for some column pair C_s and C_t from the previous step, we will now generate transformation programs using these pairs as examples. Specifically, we can view row R_s from T_s as input to a transformation program, and R_t projected on some key columns K of T_t as the desired output. If a transformation can take R_s as input and produce key columns K of R_t, equi-join becomes possible.

Physical Operators. Recall that we generate transformations using the following set of physical operators, Ω = {Split, SelectK, Concat, Substring, Constant}. The detailed interface of each operator is as follows.

- `string[] Split(string v, string sep)`
- `string SelectK(string[] array, int k)`
- `string Concat(string u, string v)`
- `string Constant(string v)`
- `string Substring(string v, int start, int length, Casing c)`

Each operator is quite self-explanatory. Split splits an input string using separator; SelectK selects the k-th element from an array; Concat performs concatenation; Constant produces a constant string; and finally Substring returns a substring from a starting index position (counting forward or backward) for a fixed length, with appropriate casing (lower case, upper case, title case, etc.).

In designing the operator space for Auto-Join, we referred

to the string transformation primitives defined in the spec of the `String` class of C# and Java. The physical operators we use in the end are a subset of the built-in `String` functions of these languages. More discussions on the choice of the operators can be found in Appendix D.

Disambiguate Transformations by Examples. While we initially illustrate transformations using one joinable row pair for simplicity in Example 1; in practice, given only one joinable row pair there often exists multiple plausible transformations.

Example 6. In Example 1 the input row denoted as X has two elements $\{[\texttt{Obama, Barack(1961-)}], [\texttt{47.0}]\}$, and the target output is $[\texttt{Barack Obama}]$. In addition to the transformation shown in that example, an alternative transformations that can also produce this output is:

1. Take the first item $X[0]$, Substr[8:6], produce $[\texttt{Barack}]$
2. Concat with a constant string " ", produce $[\texttt{Barack }]$
3. Concat again with $X[0]$, Substr[0:5], produce $[\texttt{Barack Obama}]$

There exists many candidate transformations given only one input/output example pair. However, most of transformations would fail to generalize to other example pairs. The observation here is that if we use multiple joinable row pairs as input/output examples, the space of possible transformations are significantly constrained, such that the incorrect transformations will be pruned out. For example, if we just add the second rows from Figure 1 as an example pair, with $\{[\texttt{Bush, George W.(1946-)}], [\texttt{49.4}]\}$ as the input and $[\texttt{George W. Bush}]$ as the output, then the transformation discussed in Example 6 would no longer be valid, as it would produce $[\texttt{eorge Bush,}]$, which cannot be joined with the keys in the other row.

The pruning power grows exponentially with the number of examples (details of this analysis can be found in Appendix H). In practice we just need a few examples (3 or 4) to constrain the space of candidate programs enough and generate the desired transformations.

Learning via Logical Operators. The learning problem now is to find consistent transformations for a small set of input/output example row pairs. While the execution of transformations can be decomposed into simple physical operators defined in Ω, these are too fine-grained and do not directly correspond to our logical view of the transformation steps that humans would take. For instance, in Example 1 when we use $\{[\texttt{Obama, Barack(1961-)}], [\texttt{47.0}]\}$ as input to produce $[\texttt{Barack Obama}]$ as output, humans would naturally view the required transformation as having three distinct logical steps – extract the component `Barack`, produce a space " ", extract the component `Obama`. Note that these logical operations correspond to a higher-level view that can always translate into a combination of simple physical operators – extracting the first of component `Barack` can be implemented as Split by "(" followed by Split by ",", and finally a Substring.

For the technical reason of learning programs from examples, by mimicking how humans rationalize transformations, we introduce a set of higher-level *logical operators* Θ, each of which can be written as a sequence of physical operators.

$\Theta = \{\text{Constant}, \text{Substr}, \text{SplitSubstr}, \text{SplitSplitSubstr}\}$

Unlike physical operators, each logical operator always returns a string. Each logical operator can be viewed as a "step" that contributes "progress" (partial output) to final results. It is important that logical operators all return strings, so that during automatic program generation, at each step we can decide which logical operator is more promising based on "progress". In comparison, physical operators like SelectK often need to be used in conjunction with other operators like SubStr before producing partial output, thus not directly amenable to automatic program generation.

Algorithm 2 Transformation learning by example

Require: $R = \{I^i, O^i | i \in [k]\}$ ▷ Input/output row pairs
1: **function** TryLearnTransform($R = \{I^i, O^i | i \in [k]\}$)
2: **while** true **do**
3: $\theta \leftarrow$ FindNextBestLogicalOp(R)
4: $P^i \leftarrow$ ExecuteOperator$(\theta, I^i, O^i), \forall i \in [k]$
5: $O^i_l =$ LeftRemainder$(O^i, P^i), \forall i \in [k]$
6: $\theta_l =$ TryLearnTransform$(\{I^i, O^i_l | i \in [k]\})$
7: **if** $\theta_l =$ **null then**
8: **continue**
9: $O^i_r =$ RightRemainder$(O^i, P^i), \forall i \in [k]$
10: $\theta_r =$ TryLearnTransform$(\{I^i, O^i_r | i \in [k]\})$
11: **if** $\theta_r =$ **null then**
12: **continue**
13: θ.left_child $= \theta_l$
14: θ.right_child $= \theta_r$
15: **return** θ ▷ current root node

The exact specification of each logical operator can be found in Appendix F. Here we give an example of rewriting SplitSubstr as a sequence of four operators.

```
string SplitSubstr(string[] array, int k, string sep,
int m, int start, int length, Casing c) :=
Substring(SelectK(Split(SelectK(array, k), sep), m),
start, length, c)
```

Using logical operators, we can define the transformation learning problem as follows.

Definition 4. *Transformation Learning*: Given a set of joinable row pairs $R = \{(R^i_s, R^i_t) | i \in [m]\}$ that can be viewed as input/output examples, and a predefined set of logical operations Θ, find a transformation $P = \theta_1 \cdot \theta_2 \cdot \theta_3 \cdot \ldots \theta_n$, $\theta_i \in \Theta$, such that
(1) P is consistent with all examples in R, namely, $\forall i \in [m]$, $P(R^i_s)$ can produce the projection of R^i_t on some key columns K of T_t, denoted as $\Pi_K(R^i_t)$;
(2) P has minimum-complexity, measured as the number of logical operators used, among all other transformation programs that are consistent with examples in R.

This definition is in spirit consistent with principles such as Minimum Description Length [20] or Occam's razor [15] – if there are multiple candidate transformations that can explain all given examples, we use the simplest one and that is likely correct. For instance, the transformation in Example 1 requires 3 logical operators, and there exist no other programs with lower complexity.

The learning problem can be viewed as a search problem – each logical operator produces a partial output and has a unit cost. Like shortest path algorithms, we want to reach the goal state by producing the required output strings but with the least cost.

This motivates a program learning algorithm that searches for the best program by recursively expanding a partial transformation program using the logical operator that yields the most progress. This algorithm is outlined in Algorithm 5 and works as follows. For the given set of input/output examples, it finds the best logical operator θ that produces the most progress towards the required output strings (which in this case is some key column of the output rows). We execute the operator θ and extract partial output produced from the target output. We get what remains to the left and right in the target output, denoted as O_l^i and O_r^i, respectively. This produces two new instances of the problem with $\{I^i, O_l^i | i \in [k]\}$ and $\{I^i, O_r^i | i \in [k]\}$, which have the same structure as the original $\{I^i, O^i | i \in [k]\}$. So we recurse and invoke TryLearnTransform on the two smaller problems. The resulting operators, θ_l and θ_r, if learned successfully, are added as the left child and right child of θ, until all remaining target output have been consumed. If at certain level in the hierarchy TryLearnTransform fails to find a consistent transformation, we can give up or optionally backtrack by using the next best logical operator. In practice we impose a limit τ on the number of logical operators that can be used in a program P to bound the search space (e.g., $\tau = 16$). Empirically, we found $\tau = 10$ to be sufficient to produce transformations needed to join all real scenarios encountered in our benchmark. Setting a larger τ however has little impact on efficiency, because incorrect program generation paths are terminated quickly for failing to generate new operators consistent with the set of output examples.

We use the following example to illustrate this procedure.

Example 7. From the example in Figure 1, suppose the first three rows from right/left tables are given as learning examples for input/output row pairs, respectively. To learn transformation, we use the first row $\{$[`Obama, Barack(1961-)`], [`47.0`]$\}$ as input and [`Barack Obama`] as output, and we use the remaining two rows as validations. To generate the first logical operator for this row pair, we search over operators in Θ with all possible parameters (separators for Split up to a certain length, indexes for Substring that are valid for the given string, etc.), and pick the logical operator that yields the most progress. In this case it can be verified that the operator with the most progress is SplitSplitSubstr, which selects the first input element: [X = `SelectK(input, 0)`]; split by "(" and take the first element: [Y = `SelectK(Split(X, "("), 0)`]; split again by " " and take the second element: [Z = `SelectK(Split(Y, " "), 1)`]; take substring from position 1 to the end [1:-1]: [`Substr(Z, 1, -1)`]. This operator generates `Barack` for the first row, `George W.` for the second, `Bill` for the third, with a total gain of 19 characters ($6+9+4$), and an average gain of 49% for the required outputs across three rows ($\frac{6}{12}$, $\frac{9}{14}$ and $\frac{4}{12}$, respectively).

With this first operator, the remaining required output strings to be covered are {" `Obama`", " `Bush`", " `Clinton`" }. We again search for the logical operator that yields the most progress, for which we find SplitSubstr that splits by ",", takes the first element, and returns the full string. Now the remaining output strings are {" ", " ", " " }, which can be covered by adding a Constant operator that produces a space character. Finally, by concatenating these operators, we complete a candidate transformation program that can be tested on the full input tables.

Through the following proposition, we show the success probability of transformation learning.

Proposition 3. *The learning procedure succeeds with high probability, if the transformation can be expressed using operators in Θ. The success probability is lower bounded by*

$$1 - \left(1 - \prod_{i \in [m]} \left(1 - \left(1 + (|S_I| + |S_I|^k)|S_O|^k\right) \frac{1}{|\Sigma|}^{k|S_i|}\right)\right)^T$$

where k is the number of independent examples used, T is the number of trials (each with k examples), $|S_I|$ and $|S_O|$ are the lengths of input/output examples, and $|S_i|$ is the length of the result of each intermediate step (for a logical operator).

A proof of this result can be found in Appendix H. This proposition shows that with T independent trials we can reduce the failure probability at a rate exponential in T, thus quickly improving success probability as T increases.

Ranking of Candidate Transformations. Recall in Section 3.1.4, we generate groups of joinable row pairs, based on q-gram matches between each pair of columns C_s, C_t. For each such group, we select the top-k row pairs with the highest scores (typically 1-to-1 matches), and apply transformation learning for a fixed number of times, each of which on a random subset of the selected row pairs. We execute each learnt transformation on the original input tables, and pick the one that joins the most number of rows in the target table. By the definition of transformation join problem (Definition 1), the joining columns in the target table are key columns (1:1 or N:1 joins). A key-column join with high row coverage is likely to be meaningful in practice. Pseudocode of this step can be found in Appendix E.2.

3.3 Join with Composite Key Columns

There exist cases where a key column in the target table is a *composite* column that joins with more than one columns in the source table. The right table in Figure 1 is such an example that has both the president names and their life spans. If we had used this composite column as the target, we would have failed to find any transformation, because the life span is missing from the left table.

While changing the direction of join in the example above would work, when both source and target key columns are composite, then neither direction would work. We use existing methods [10, 13] to split a composite columns by aligning substrings into multiple parts across all rows. For instance, one can split the key column in the right table in Figure 1 into three columns: the last name part before ",", the first name part before " (", and the life span. Since composite columns usually have strong structural regularities (e.g., punctuations) this technique produces high quality splits in most cases, from which we can apply the Auto-Join again.

To summarize, we first attempt transformation joins in two directions. When neither direction results in a transformation, we can split the key columns of input tables before applying transformation join for the second time.

4. SCALABLE AUTO-JOIN

Auto-Join is used as a data exploration feature in spreadsheet environment such as Excel, where interactivity is critical. As a result, we need to efficiently scale it to tables with thousands or even millions of rows. In this section, we explain how to achieve such scalability for Auto-Join.

Given two tables T_s and T_t, each with millions of rows, with commodity hardware it is unlikely that we can achieve interactive speed if all values need to be indexed and probed. On the other hand, it would actually be wasteful to index/probe all values, because for the purpose of transformation learning we only need enough joinable row pairs for learning to be successful. Intuitively, we can sample rows from input tables, where the key is to use appropriate sampling rates to minimize processing costs but still guarantee success with high probability (the danger is that we may under-sample, thus missing join relationships that exist).

For Auto-Join, we use *independent row samples* from input tables. Unlike sampling techniques for equi-join or set-intersection, where *co-ordinated sampling* can be used as join keys are explicitly given, in our problem, the join keys are not known *a priori* and we have to rely on independent samples. Although q-gram sampling is a possible alternative, the cost of analyzing and hashing all q-grams can be prohibitive for large tables. With these considerations, we now study a lightweight row sampling for Auto-Join.

Let N_s, N_t be the number of rows in table T_s, T_t, and p_s, p_t be their sampling rates, respectively. Furthermore, let r be the *join participation rate*, defined as the fraction of records in the target table T_t that participate in the desired join. Note that the join participation rate needs to be reasonably high for sampling to succeed – if only one row in a million-row table participates in a join, then we need to sample a very large fraction of it to find the joinable row pair with high probability. In practice, r is likely to be high, because joins that humans find interesting likely involve a non-trivial fraction of rows. We conservatively set r to a low value (e.g., 1%), so that as long as the real join participation is higher we can succeed with high probability.

We formulate the problem of determining p_s and p_t as an optimization problem. The objective is to minimize the total number of sampled rows that need to be indexed and queried, which is $N_s p_s + N_t p_t$. The constraint is that we need to sample enough joinable row pairs with high probability. Since the join participation rate is r, at least $N_t p_t r$ rows from the target table T_t participate in join. Because each of these participating row joins with at least one row from the source table T_s, which is sampled with probability p_s, leading to an expectation of $\mu = N_t \cdot p_t \cdot r \cdot p_s$ joinable row pairs in the sample. This can be seen as a Bernoulli process with a success probability of $p_t p_s r$ and N_t total trials.

As discussed in Section 3.2, we need a certain number of examples to constrain the space of feasible transformations enough to produce correct transformations. Let this required number be T (empirically 4 is enough in most cases). Let X be a random variable to denote the total number of joinable row pairs sampled. We want to find an upperbound for the probability that less than T joinable rows pairs are sampled, or $P(X \leq T)$.

Using the Multiplicative Chernoff Bound [9], we know X can be bounded by

$$P(X \leq (1-\delta)\mu) \leq e^{-\frac{\delta^2 \mu}{2}} \tag{5}$$

If we have $\mu \geq \frac{T}{1-\delta}$, we can upper-bound the failure probability as

$$P(X \leq T) \leq e^{-\frac{\delta^2 \mu}{2}} \tag{6}$$

For example, let $T = 4$, $\delta = 0.8$, we get $\mu = N_t p_t p_s r > \frac{T}{1-\delta} = 20$. Using Equation 6, we get $P(X \leq T) \leq e^{-\frac{0.64 \cdot 20}{2}} = e^{-6.4} < 0.0017$, or in other words, our success probability is at least 99.8%. So as long as we can ensure $\mu = N_t p_t p_s r > \frac{T}{1-\delta}$, then more than T joinable row pairs will be sampled with high probability. This becomes the constraint that completes our optimization problem:

$$\begin{aligned} \min \quad & N_s p_s + N_t p_t \\ \text{s.t.} \quad & N_t p_t p_s r \geq \frac{T}{1-\delta},\ p_t, p_s \in [0,1] \end{aligned} \tag{7}$$

Using Lagrange we obtain the following closed form optimal solution [3]:

$$p_t = \sqrt{\frac{T}{(1-\delta) r N_s}},\ p_s = \sqrt{\frac{T N_s}{(1-\delta) r N_t^2}} \tag{8}$$

The corresponding sample sizes can be written as $N_t p_t = \frac{N_t}{N_s}\sqrt{\frac{T N_s}{(1-\delta)r}}$ and $N_s p_s = \frac{N_s}{N_t}\sqrt{\frac{T N_s}{(1-\delta)r}}$, and both of them grow sub-linearly in N_s.

As a concrete example, suppose we have two tables both with 1M rows. For some fixed setting of T and δ, such as $T = 4$ and $\delta = 0.8$ from the previous example that guarantees success with high probability, and $r = 0.1$, we can compute the sampling rates as $p_t = 0.014$ and $p_s = 0.014$, which translates to a small sample of 14K rows from the 1M-row tables.

Using the optimized sampling technique, we significantly reduce the processing cost and improve the efficiency of Auto-Join in dealing with large tables.

5. CONSTRAINED FUZZY JOIN

For datasets from the Web, inconsistencies are often found in how values are formatting and represented (e.g., with or without middle name and middle initials for names). Thus, an equi-join using transformation may miss some row pairs that should join, as Example 8 shows.

EXAMPLE 8. In Figure 2, the learned transformation that has the highest coverage on the target key column `Email` concatenates the first character of the first name with the last name in the left table to get the email addresses. However this transformation does not cover the target key `mi-payne@forsyth.k12.ga.us` as it uses the first two characters in the first name. As a result, the third rows in the left and right tables cannot be equi-joined.

Traditionally fuzzy join is used to deal with small value inconsistencies. However, given a wide space of parameters in fuzzy join such as the tokenization, distance function, and threshold, configuring a fuzzy join that works well for a given problem instance is difficult. This is particularly true for Auto-Join, as it is intended to be a data exploration feature in spreadsheets where users may not have no the expertise on fuzzy joins.

We propose to automatically optimize a fuzzy join configuration using rows that are already equi-joinable as constraints, so that as we relax matching criteria in fuzzy join, we do not make these rows to join more than their equi-join results (which indicates that the corresponding fuzzy join is too lax). Although we use this optimization in the context

[3]When N_s and N_t are small, there may not be feasible solutions (the required p may be greater than 1). In such cases we use full tables.

of Auto-Join, the techniques here are of independent interest and can be used to optimize general fuzzy join.

Given a column C produced by transformations on the source table, and K a key column from the target table, where the join is 1:1 or N:1, our optimization objective is to maximize the number of rows in the target table that can be fuzzy-joined between C and K, subjecting to the constraint on join cardinality.

Specifically, given a tokenziation scheme t from a space of possible configurations (e.g., word, 2-gram, 3-gram, etc.), a distance function d (e.g., Jaccard, Cosine), and a distance threshold (normalized into a fixed range, e.g., [0,1]). The rows that can fuzzy join for some given t, d, s, denoted as $F_{t,d,s}(C, K)$, is defined as follow.

$$F_{t,d,s}(C,K) = \{v_k \mid \exists v_k \in K, v_c \in C, d_t(v_k, v_c) \leq s\} \quad (9)$$

where d_t is the distance d using a given tokenization t.

This objective alone tends to produce overly lax matches. The counteracting constraint is to respect join cardinality. Specifically, after using fuzzy join every value $v_c \in C$ cannot join with more than one value $v_k \in K$. This can be viewed as a key-foreign-key join constraint – a foreign-key value should not join with two key values (even with fuzzy join).

Additionally, we can optionally require that each $v_k \in K$ cannot join with more than one distinct $v_c \in C$. This is an optional constraint assuming that on the foreign key side, each entity is only represented with one distinct value. E.g., if we already have "`George W. Bush`" in a table, we would not have "`George Bush`" or "`George W. Bush Jr.`" for the same person. On the other hand a very close value "`George H. W. Bush`" in the same column likely corresponds to a different entity and should not join with the same key as "`George W. Bush`". This optional requirement helps to ensure high precision.

These requirements lead to the following two constraints in the optimization problem.

$$\begin{aligned} &\arg\max_{t,d,s} |F_{t,d,s}(C,K)| \\ &\quad s.t.\ |\{v_k \mid v_k \in K, d_t(v_c, v_k) \leq s\}| \leq 1, \forall v_c \in C \\ &\quad\quad |\{v_c \mid v_c \in C, d_t(v_c, v_k) \leq s\}| \leq 1, \forall v_k \in K \end{aligned} \quad (10)$$

We can search over possible t, d, and s from a given parameterization space. The following example illustrates how fuzzy join optimzation is used for joining tables in Figure 1.

Example 9. Continue with Example 8, after applying the transformation, the output for `Missy Payne` in the left table is `mpayne@forsyth.k12.ga.us`, which cannot be equi-joined with `mipayne@forsyth.k12.ga.us` in the right table. Using 3-gram tokenizer and Jaccard distance, the distance between the two is 0.125. Thus, a distance threshold above 0.125 would join these two rows. On the other hand, if we use a larger distance threshold such as 0.4, `kmoore@forsyth.k12.ga.us` (transformation output from the left table) would join `mipayne@forsyth.k12.ga.us` (in the right table), breaking the second constraint in Equation 10 as `mipayne@forsyth.k12.ga.us` is joined with two distinct values. The fuzzy join optimization algorithm finds the maximum threshold that still satisfies the join constraints, thus the optimal threshold in this case is 0.2 or 0.3.

Due to the monotonicity property of the objective function with respect to the distance threshold, we use binary search to find the optimal distance threshold. The pseudo code for fuzzy join optimzation can be found in Appendix E.4.

6. EXPERIMENTS

In this section we discuss experimental results on join quality (precision/recall) as well as scalability.

6.1 Benchmark Datasets

Benchmarks. We constructed two benchmark, `Web` and `Enterprise`, using test cases from real datasets; as well as a synthetic benchmark `Synthetic`. The `Web` benchmarks is published online[4] to facilitate future research.

The `Web` benchmark is constructed using tables on the Web. We sampled table-intent queries from the Bing search engine that have the prefix "list of" (e.g., "list of US presidents"). We then used Google Tables [2] to find a list of tables for that query (e.g., U.S. presidents), and selected pairs of tables that use different representations but are still joinable under transformation (e.g., Figure 1). We searched 17 topics and collected 31 table pairs. We observe that tables on the Web often have minor inconsistencies (e.g., formatting differences, with or without middle initials in names, etc.) even when they are on the same topic, and many such inconsistencies cannot be easily accounted for using transformations alone. This makes `Web` a difficult benchmark.

The `Enterprise` benchmark contains 38 test cases, each of which has a pair of tables extracted from spreadsheet files found in the intranet of a large enterprise (e.g., Figure 3 and Figure 4). The test cases are constructed by grouping tables with common topics. Comparing to `Web` that has mostly 1-to-1, entity-to-entity joins, `Enterprise` also has cases with hierarchical N:1 joins (e.g., Figure 3).

Lastly, since the authors in [22] studied a close variant of the auto-join problem and used synthetic datasets for evaluation, as a validation test we also reconstruct 4 synthetic datasets used in [22] to create the `Synthetic` benchmark. The 4 test cases, `UserID`, `Time`, `NameConcat`, and `Citeseer`, are constructed by either splitting or merging columns from source tables to produce target tables [22].

In all these benchmark cases, equi-join would fail. We manually created the ground truth join result for each pair of tables, by determining what rows in one table should join with what rows from the other table.

Evaluation metrics. We use the following metrics to measure join quality. Denote by G the row pairs in the ground truth join results, J the joined row pairs produced by an algorithm. We measure join quality using the standard *precision* and *recall*, defined as:

$$precision = \frac{|G \cap J|}{|J|},\ recall = \frac{|G \cap J|}{|G|}$$

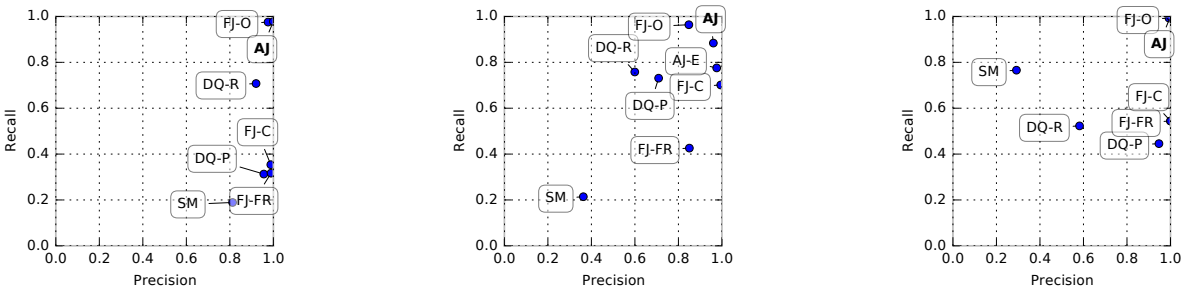
We also report *F-score* that is the harmonic mean of precision and recall. When an algorithm produces empty join results, we do not include it in computing average precision, but we include it in average recall and F-score.

6.2 Methods Compared

We implemented 8 methods for comparison.

Substring Matching (`SM`). We implemented the algorithm by Warren and Tompa based on their paper [22]. This

[4]https://www.microsoft.com/en-us/research/publication/auto-join-joining-tables-leveraging-transformations/



(a) Enterprise Benchmark (b) Web Table Benchmark (c) Synthetic Benchmark

Figure 5: Average precision/recall comparison for all methods on three benchmarks.

algorithm uses a greedy strategy to find a *translation formula*, which is a sequence of indexes of the source columns' substrings that matches parts of the target column. The translation formula is constructed incrementally by inspecting one source column at a time with no reverse-back. When inspecting each source column, `SM` finds the source column's substring indexes that lead to the highest number of successful alignments with the target column, and then uses the substring indexes to form the partial translation formula, which is added to the main formula.

Fuzzy Join - Oracle (`FJ-O`). It is known that a major difficulty of using fuzzy join is the need to find the right configuration from a large space of parameters, which includes different tokenization schemes, distance functions, and distance thresholds, etc. To be more favorable to fuzzy join based methods, we consider an extensive combination of configurations. Specifically, for tokenization we use {Exact, Lower, Split, Word, and q-gram for $q \in [2, 10]$} (similar to ones used in [17]); for distance functions we consider {Intersect, Jaccard, Dice, MaxInclusion}; and for thresholds we use 10 equally-distanced values (e.g., {0.1, 0.2, ..., 1} for Jaccard). This creates a total of 520 unique parameter configurations. We execute each of these fuzzy joins on columns that are used in the ground truth as if they are known *a priori*, and we join each row with top-1 fuzzy match in the other table to maintain high precision. We report the best configuration that has the highest average F-score across all cases.

Note that this method acts much like an "Oracle" – it has access to not only the columns that join, but also the ground truth join result to "fine tune" its configuration for the best performance. These optimizations are not feasible in practice, so this provides an upper bound on what fuzzy join like methods can achieve.

Fuzzy Join - Column (`FJ-C`). In this method, we perform fuzzy join on columns that participate in joins in the ground truth as if these are known, but without using detailed row-level ground truth of which rows should join with which for configuration optimization. We use techniques discussed in Section 5 to determine the best parameter configuration.

Fuzzy Join - Full Row (`FJ-FR`). This fuzzy join variant is similar to `FJ-C`, but we do not provide the information on which columns are used in join in the ground truth. As a result, this approach considers full rows in each table. This represents a realistic scenario of how fuzzy join would be used without ground truth.

Dynamic q-gram - Precision (`DQ-P`). Since q-grams already identify some joinable row-pairs (from which we generate transformations), one may wonder if it is sufficient to perform join using q-gram matches alone. In this method we use matches produced in Section 3.1, and only allow 1-to-1 q-gram matches to ensure high precision. Joinable row pairs are used directly as join result.

Dynamic q-gram - Recall (`DQ-R`). This algorithm is similar to `DQ-P`, except that we allow n-to-1 q-gram matches as join results. This produces results of higher recall but can also lead to lower precision compared to `DQ-P`.

Auto-Join (`AJ`). This is our Auto-Join algorithm. We create a variant **Auto-Join - Equality** (shorten as `AJ-E`) that only uses equality join without the fuzzy join described in Section 5.

6.3 Quality Comparison

In this section we discuss the experimental results on three benchmarks. Figure 5 shows the average precisions and recalls comparison on all three benchmarks. Table 2, Figure 6 and 7 show the F-scores on all the benchmark datasets.

6.3.1 *Enterprise Benchmark*

`Enterprise` contains data that are mostly clean and well structured – values are consistently encoded with few typos or inconsistencies as they are likely dumped from sources like relational databases (e.g., Figure 3 and Figure 4). Unlike other benchmarks, a significant fraction of joins are N:1 join through hierarchically relationships (e.g., Figure 3).

The precision and recall results are show in Figure 9a. First, Auto-Join (`AJ`) achieves near-perfect precision (0.9997) and recall (0.9781) on average. In comparison, the oracle baseline `FJ-O` has precision at 0.9756 and recall at 0.9755, which is inferior to Auto-Join. Recall that `FJ-O` is the Oracle version of fuzzy join that uses ground truth to find the best possible configuration, which provides an upper-bound for fuzzy join and not feasible in practice. This demonstrates the advantage of transformation-based join over fuzzy join when consistent transformations exist. We note that in this test case using equality-join only `AJ-E` (with no optimized fuzzy join) produces virtually the same quality results, because values in this benchmark are clean and well structured.

Second, the `SM` algorithm achieves lower precision and recall then other baselines methods. This shows that their approach in building the translation formula using fixed substring indexes, as mentioned earlier in Section 6.2, is not expressive enough to handle transformations needed in real join scenarios we encountered.

Third, fuzzy join algorithms (`FJ-FR` and `FJ-C`) produced

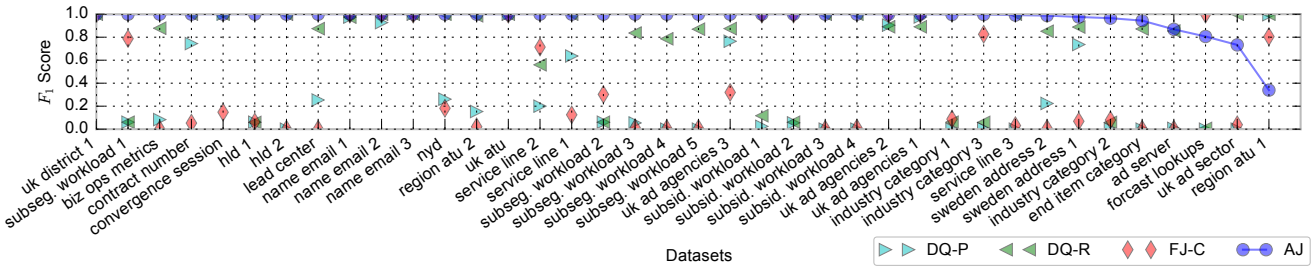


Figure 6: F-Scores on Enterprise Benchmark

good precision due to our conservative fuzzy-join optimization. However, their recall is low, because in certain cases where the join values are hierarchical, non-joinable row pairs may also have low syntactic distance (e.g., Figure 3), which makes it difficult to differentiate between joinable/non-joinable row pairs using distance functions alone.

Lastly, `DQ-P` produces joins based on 1-to-1 q-gram matches, which has high precision but low recall. This is consistent with our analysis that 1-to-1 q-gram matches are often good joinable row pairs for transformation learning. On the other hand, `DQ-R` relaxes the matching constraints, and as expected produces better recall but lower precision.

Figure 6 shows the F-scores on individual cases. It is clear that in most datasets, `AJ` achieved higher scores than the baselines, demonstrating that `AJ` is more resilient to complications such as N:1 joins and common substrings between joinable and non-joinable row pairs. In the test cases `uk ad sector` and `region atu 1`, `AJ` did worse than `DQ` methods. A close inspection reveals that it finds an alternative transformation that only covers a subset of the joinable results.

6.3.2 Web Benchmark

Unlike `Enterprise` that have a significant number of N:1 joins, in `Web` most join cases are 1:1, entity-to-entity joins (e.g., Figure 1 and Figure 2), and are considerably more dirty with ad-hoc inconsistencies.

Figure 9b gives the quality comparisons. First, `AJ` has a considerably higher average precision than the oracle fuzzy join `FJ-O`, but a lower average recall. This is not surprising because `FJ-O` uses ground truth to optimizing its configuration parameters, which is not feasible in practice. We do notice that because `FJ-O` always joins a row with its top-1 match by score as long as the score is above a certain threshold, which leads to many false positives and thus lower precision. The problem is the most apparent for cases where most rows from one table do not actually participate in join. This is an inherent shortcoming of top-1 fuzzy join methods that `AJ` can overcome.

We see in Figure 9b that Auto-Join (`AJ`) has a higher average recall than its equality join version, `AJ-E` (0.8840 vs. 0.7757), but slightly lower precision (0.9504 vs. 0.9758). This is because inconsistencies exist in certain cases, where one correct transformation alone does apply to all rows that should join. In such cases, optimized fuzzy join brings significant gain in recall (≈ 0.11), with a small loss in precision (≈ 0.025).

`SM` does not perform well compared to other methods. Transformations required in `Web` benchmark are often too more complex for `SM` that relies on fixed substring indexes.

We analyze individual cases for which `FJ-C` produces higher F-scores than `AJ`, as shown in Figure 7. For cases like `uk pms`, we found that although `AJ` learnt the correct transformation and achieved a perfect precision, the fuzzy join step was not able to cover the rest of joinable row pairs that have inconsistencies in entities' naming. For complex cases like `duke cs profs`, the correct join actually requires more than one transformations in order to join all rows. Although `AJ` learns one transformation with perfect precision, it falls short in recall as not all joinable rows are covered. For these two datasets, the fuzzy distances between the non-joinable row pairs using the original join-columns are larger than when using the derived column. So it is easier for `FJ-C`, which uses the original join-columns, to differentiate between joinable and non-joinable row pairs and achieve higher F-score, even though `FJ-C` and `AJ` uses the same fuzzy join method.

	Citeseer	NameConcat	Time	UserID
DQ-P	0.9826	0.1264	0.0392	0.7572
DQ-R	0.9826	0.1356	0.2025	0.6638
FJ-C	0.4637	0.1651	1.0000	0.8795
SM	0.0291	0.1186	0.5464	0.7553
AJ	1.0000	1.0000	1.0000	1.0000

Table 2: F-Scores on Synthetic Benchmark

6.3.3 Synthetic Benchmark

`Synthetic` contains cases synthetically generated as described in prior work [22] using split or concatenation. The cases here are relatively simple and we use these as a validation test to complement with our previous benchmarks.

Figure 9d shows that `AJ` achieves perfect precision and recall, matching the oracle fuzzy join `FJ-O`. Other methods produce results similar to the previous benchmarks.

Table 2 shows the F-scores on individual cases. Both of `DQ-P` and `DQ-R` performs poorly on the `Time` dataset. `Time` is synthetically generated by concatenating three columns with second (0-59), minute (0-59), and hour (0-23) into a time column separated by ":". Due to the small space of numbers, there are many common substrings and few 1-to-1 or n-to-1 q-gram matches, thus the low scores of `DQ-P` and `DQ-R`. These two approaches work well on `Citeseer`, which has many 1-to-1 q-gram matches due to unique author names and publication titles. `AJ` achieved perfect F-scores on all datasets, since it just needs a few examples to produce the generalized transformations needed.

We found that `SM` achieved good recall in this benchmark, however, its average precision is relatively low (see Figure 9d). This result is not as well as what is reported in the original work [22]. This is likely because the method is

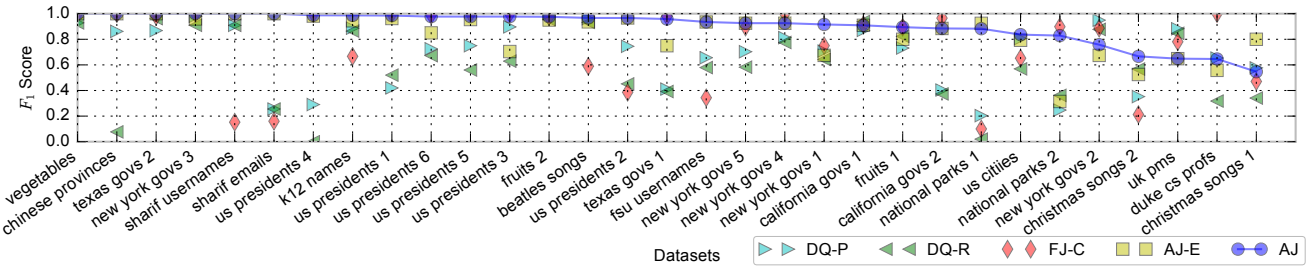


Figure 7: F-Scores on Web Table Benchmark

data-sensitive, and it tends to fall into a local optimum with its use of greedy strategy in finding a translation formula. Since the translation formula is constructed incrementally by inspecting one source column at a time with no reverse-back, the addition of a single incorrect partial formula stops the whole algorithm from finding the globally optimal formula. This step is quite sensitive to the variance in the lengths of the substrings that matches with the target column. This is evident in Table 2, as `SM` did relatively better in `Time` and `UserID`, which has smaller variances (`Time` has zero variance, and `UserID` uses a fixed-length substring in the translation formula), while the scores in `Citeseer` and `NameConcat` are much lower.

6.4 Scalability Evaluation

We used the DBLP datasets [1] to evaluate the scalability of `SM`, `FJ-O`, `FJ-C`, and `AJ`. In the DBLP data set, each record has three fields: authors, title, and year of publication. For the purpose of scalability evaluation, we create a target table that is the concatenation of these three fields. We sample N records from the source table and the target table where N = {100, 1K, 10K, 100K, 1M}, and measure the corresponding end-to-end execution time. Some existing methods are very slow on large data sets so we set a timeout at 2 hours. Note that we omit results for `FJ-FR` since it is identical to `FJ-C` for this data set. We also do not compare with `DQ-R` and `DQ-P` since these are sub-components from the proposed `AJ` method.

Figure 8 shows the end-to-end running times. `AJ` is 2-3 orders of magnitude faster than existing methods. In particular, `SM` and `FJ-O` time out at 10K rows, and `FJ-C` times out at 100K rows. The runtime of these methods grow quickly with the table sizes.

For `AJ`, we break down the time into three stages. The first stage is the creation of the suffix array indexes (Indexing). The second stage is finding joinable row pair and transformation learning (Find Trans). The third stage is to perform equality join (Equi-Join). The indexing time gradually dominates as the number of rows grows over 100K. On the other hand, the cost of transformation learning increases very slowly, from 2.5 seconds at 100 rows to 6.4 seconds at 1M rows, since the number of attempts for transformations learning does not increase as the table grows.

In addition, we experimented Auto-Join without the optimized row sampling (Section 4) to see its impact on efficiency. Without sampling the algorithm reaches timeout on the data set with 1M rows, and is more than 5 times slower on the 100K data set. The sampling-based optimization is clearly important to scale to large data sets.

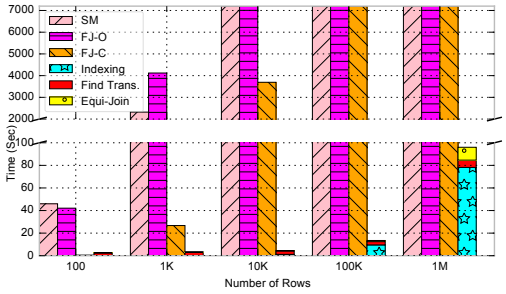


Figure 8: End-to-end running time comparison

7. RELATED WORKS

Warren and Tompa proposed a schema translation technique in the context of data integration [22], which is closely related to Auto-Join. Their technique is based on string alignment that produces a transformation given two tables as input. However, the set of transformations they consider is limited to only concatenation, which is not sufficient to handle transformations required to join real tables as our experiment results show. Our approach, in comparison, supports much more expressive transformations that can handle almost all real join cases encountered, provides probabilistic guarantee of its success, and maintain an interactive speed even for large tables.

There is a long and fruitful line of research on schema matching that identifies column correspondence between tables, where certain types of transformations have been considered. For example, *iMap* [12] was developed to handle transformations such as numerical conversion and string concatenation. Similarly *Bellman* [11] can also find column correspondence when the underlying transformation is simple string concatenations.

Compared to Auto-Join, in schema matching one only needs to identify column-level correspondence for humans to judge, where no precise correspondence is produced at the row-level. In Auto-Join we need to reproduce the underlying transformation at the row-level in a generative process. Furthermore in schema matching transformations considered are limited since q-gram match and fuzzy matching is often sufficient to identify column correspondence. Techniques we develop for Auto-Join can precisely identify complex relationships between columns, which can in fact be used for schema matching (with high confidence).

Program-by-example [18] is a programming paradigm studied in the programming language community to facilitate program generation based on examples, which is closely re-

lated to our transformation learning component. Systems such as FlashFill [16] and BlinkFill [21] perform transformations when given input/output examples. In comparison, our system automatically identifies such examples. Our program generation algorithm is also considerably different from existing techniques that requires efficient enumeration of all feasible programs.

8. CONCLUSION

We developed Auto-Join, a system for automatically join tables with transformations. We show that Auto-Join is able to produce high-quality join result that significantly outperform existing solutions. An interesting future direction is to look at ways to automatically join tables with semantic relationships.

9. REFERENCES

[1] DBLP. http://dblp.uni-trier.de/.

[2] Google Web Tables. http://research.google.com/tables.

[3] Informatica rev. https://www.informatica.com/products/data-quality/rev.html.

[4] Microsoft Excel Power Query. http://office.microsoft.com/powerbi.

[5] Power query: Merge queries. https://support.office.com/en-us/article/Merge-queries-Power-Query-fd157620-5470-4c0f-b132-7ca2616d17f9.

[6] C. Biemann. *Structure Discovery in Natural Language*. Theory and Applications of Natural Language Processing. Springer, 2012.

[7] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. In *Computer Networks and ISDN Systems*, 1997.

[8] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *Proceedings of ICDE*, 2006.

[9] H. Chernoff. A note on an inequality involving the normal distribution. *Annals of Probability*, 1981.

[10] X. Chu, Y. He, K. Chakrabarti, and K. Ganjam. Tegra: Table extraction by global record alignment. In *Proceedings of SIGMOD*, 2015.

[11] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *SIGMOD*, pages 240–251, 2002.

[12] R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. imap: Discovering complex semantic matches between database schemas. In *SIGMOD*, SIGMOD '04, pages 383–394, New York, NY, USA, 2004. ACM.

[13] H. Elmeleegy, J. Madhavan, and A. Halevy. Harvesting relational tables from lists on the web. *The VLDB Journal*, 20(2):209–226, Apr. 2011.

[14] N. Ganguly, A. Deutsch, and A. Mukherjee. Dynamics on and of complex networks: Applications to biology, computer science, and the social sciences. 2009.

[15] H. G. Gauch. *Scientific Method in Practice*. Cambridge University Press, 2003.

[16] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *Proceedings of SIGPLAN*, pages 317–328, 2011.

[17] O. Hassanzadeh, K. Q. Pu, S. H. Yeganeh, R. J. Miller, L. Popa, M. A. Hernández, and H. Ho. Discovering linkage points over web data. *PVLDB*, 6(6):444–456, 2013.

[18] H. Lieberman, editor. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann, 2001.

[19] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[20] J. Rissanen. Modeling by shortest data description. *Automatica*, 1978.

[21] R. Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. In *Proc. VLDB*, 2016.

[22] R. H. Warren and F. W. Tompa. Multi-column substring matching for database schema translation. In *PVLDB*, pages 331–342, 2006.

APPENDIX

A. SUFFIX ARRAY INDEX

Here we present a simple example of a suffix array index. As we have discussed in Section 3.1.3, we use suffix array indexes for finding q-gram matches: given a query q-gram from a source column, we want to find a value in a target column that contains the q-gram exactly.

In this example, we match a person's full-name with its email address. For the sake of simplicity, let the target column contain only one value, `surajitc@microsoft.com`. To build a suffix array index, we first extract all the suffixes of the value as follows:

```
["surajitc@microsoft.com",
 "urajitc@microsoft.com$",
 "rajitc@microsoft.com$$",
 "ajitc@microsoft.com$$$",
 "jitc@microsoft.com$$$$",
 "itc@microsoft.com$$$$$",
 "tc@microsoft.com$$$$$$",
 "c@microsoft.com$$$$$$$",
 "@microsoft.com$$$$$$$$",
 "microsoft.com$$$$$$$$$",
 "icrosoft.com$$$$$$$$$$",
 "crosoft.com$$$$$$$$$$$",
 "rosoft.com$$$$$$$$$$$$",
 "osoft.com$$$$$$$$$$$$$",
 "soft.com$$$$$$$$$$$$$$",
 "oft.com$$$$$$$$$$$$$$$",
 "ft.com$$$$$$$$$$$$$$$$",
 "t.com$$$$$$$$$$$$$$$$$",
 ".com$$$$$$$$$$$$$$$$$$",
 "com$$$$$$$$$$$$$$$$$$$",
 "om$$$$$$$$$$$$$$$$$$$$",
 "m$$$$$$$$$$$$$$$$$$$$$"]
```

Note we pad the suffixes with `$`. Then we sort the suffixes in ascending order:

```
[".com$$$$$$$$$$$$$$$$$$",
 "@microsoft.com$$$$$$$$",
 "ajitc@microsoft.com$$$",
 "c@microsoft.com$$$$$$$",
 "com$$$$$$$$$$$$$$$$$$$",
 "crosoft.com$$$$$$$$$$$",
 "ft.com$$$$$$$$$$$$$$$$",
 "icrosoft.com$$$$$$$$$$",
 "itc@microsoft.com$$$$$",
 "jitc@microsoft.com$$$$",
 "m$$$$$$$$$$$$$$$$$$$$$",
 "microsoft.com$$$$$$$$$",
 "oft.com$$$$$$$$$$$$$$$",
 "om$$$$$$$$$$$$$$$$$$$$",
 "osoft.com$$$$$$$$$$$$$",
 "rajitc@microsoft.com$$",
 "rosoft.com$$$$$$$$$$$$",
 "soft.com$$$$$$$$$$$$$$",
 "surajitc@microsoft.com",
 "t.com$$$$$$$$$$$$$$$$$",
 "tc@microsoft.com$$$$$$",
 "urajitc@microsoft.com$"]
```

Let the query be `urajit`, a 6-gram from the name `Surajit Chaudhuri`. We want to find a suffix in the array that has an exact *prefix match* with the query q-gram. Since the array is sorted, we can use binary search. In this case, the matching suffix is `urajitc@microsoft.com$`, whose prefix of length 6 matches the query q-gram exactly. The search complexity is $O(\log(S))$ where S is the total number of suffixes.

B. POWER LAW IN TABLE CORPUS

We have conducted experiments that count q-gram occurrences from over 100M Web tables extracted from a recent snapshot of the documents indexed by the Bing search engine. We test whether q-grams and natural words in the large table corpus also follow power laws by plotting a rank vs. frequency graph as shown in Figure 9. Note that both axes of the plots are in log scale. The close-to-linear relationships observed on these plots (especially for words and q-grams with larger q) suggest that this is indeed the case. This provides justification for our q-gram-probing based design that leverages unique q-gram matches to find joinable row pairs.

C. PROOFS OF PROPOSITIONS

Proof of Proposition 1. Given Equation 2, the probability that a q-gram with rank k appears exactly once in a random collection of N q-grams follows a geometric distribution:

$$(1 - p_q(k))^{N-1} \cdot p_q(k)$$

The probability of this q-gram appears exactly once each in two independent collections with N q-grams is then:

$$\Big((1 - p_q(k))^{N-1} \cdot p_q(k)\Big)^2$$

Lastly, the probability that at least one q-gram appears exactly once each in two collections can be computed using a summation over all possible $|\Sigma|^q$ q-grams:

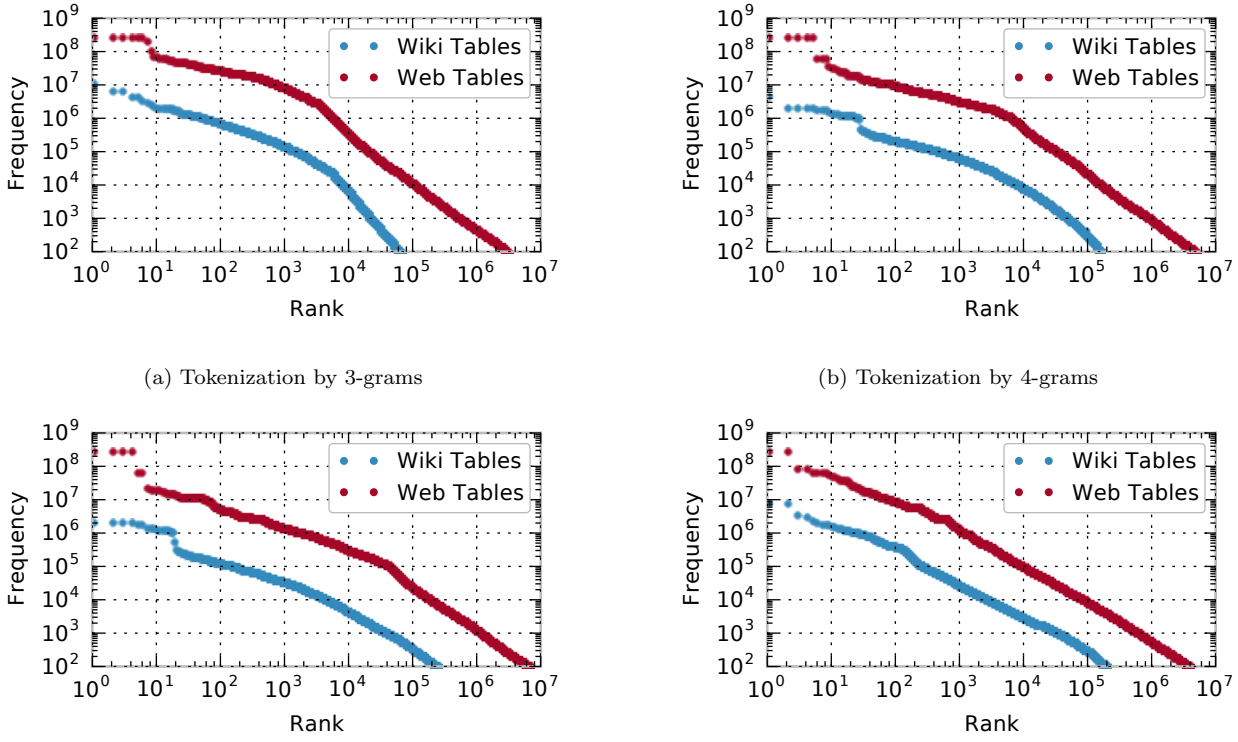
$$\sum_{k=1}^{|\Sigma|^q} \Big((1 - p_q(k))^{N-1} \cdot p_q(k)\Big)^2$$

□

Proof of Proposition 2. This proof is for the monotonicity of the number of q-gram matches with respect to q, which in the context of Proposition 2 is the length of the prefix g_u^q of the suffix u. Let a $g_u^q = x_1 x_2 \dots x_q$ be a prefix with length q, and $S = s_1 s_2 \dots s_n$ be a string of length n, where $x_1, x_2, \dots$ and $s_1, s_2, \dots$ are characters. Let $S[i:j]$ be a slice of S from s_i to s_j, where $1 \le i < j \le n$. If $S[i:j] = g_u^q$, then $S[i:j-1] = g_u^q[1:q-1]$. Thus, if g_u^q has one match with S, then $g_u^q[1:q-1]$ must have at least one match with S. Reversely, if $S[i:j-1] = g_u^q[1:q-1]$ but $s_j \neq a_q$, then $S[i:j] \neq g_u^q$. Thus, if $g_u^q[1:q-1]$ has a match with S, then g_u^q may not have a match with S. Therefore, the number of matches for a prefix g_u^q with a string S monotonically decreases as q increases. □

D. DESIGN OF Auto-Join OPERATORS

We believe that for a good design Auto-Join operators need to be (1) expressive enough to support all practical join cases; and (2) amenable to human understanding and algorithmic inference. The criterion (1) is easy to understand – we want to have a language that covers most if not all cases that require transformation-based joins. This alone



(a) Tokenization by 3-grams

(b) Tokenization by 4-grams

(c) Tokenization by 5-grams

(d) Tokenization by words

Figure 9: Frequency vs. rank in log-log plots. Close-to-linear relationships in the plots show that q-grams in Web tables and Wikipedia tables also follow power laws, similar to what people have observed in general natural language text [6, 14].

however is not enough, because a naive design to satisfy this requirement is to introduce one operator specifically for each join case, which would create a bloated set of operators with partially overlapping semantics. The criterion (2) is important in this context – a natural and succinct language will be easier for humans to understand/debug (advanced users may want to inspect join-programs to ensure correctness). It is also important for the technical reason of learnability, because a succinct set of operators with non-overlapping semantics is easier to learn from examples (a language with partially overlapping operators, on the other hand, will create parallel versions of intermediate programs that are expensive to converge).

In designing the operators for Auto-Join, we referred to the string transformation primitives defined in the spec of the `String` class of C# [5] and Java [6]. As described in Section 3.2, the physical operators we use in the end are {Concat, Substr, Split, SelectK, Constant}, which is really a subset of the built-in `String` functions in C# and Java. These languages are time-tested, which are clearly expressive and natural to humans. The primitives are known to have disjoint in semantics that are great for composability. By adopting these operators, our language should inherit the good properties discussed above.

To validate that the operators used in auto-join are indeed both natural and expressive for real join scenarios, we conduct the following exercise. We ask a human expert to manually label all benchmark cases using any natural string transformation primitives that he can think of. The human expert is able to write ground-truth programs for all but 4 difficult test cases for which consistent transformation programs do not exist. We will get to these cases shortly. For the remaining cases where the human expert is able to write ground-truth programs, we count the number of occurrences of each operator used in the ground truth. It turns out that all the operators in the ground truth can be mapped to the operators we use in auto-join – a detailed breakdown of operator distribution is shown in Figure 10. This result suggests that the operators in auto-join are not only expressive for real-world join cases, but also natural to humans – if one would have to manually write transformation-join programs, they would be the same as the ones we automatically generate.

[5] C# String class: https://msdn.microsoft.com/en-us/library/system.string(v=vs.110).aspx

[6] Java String class: https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

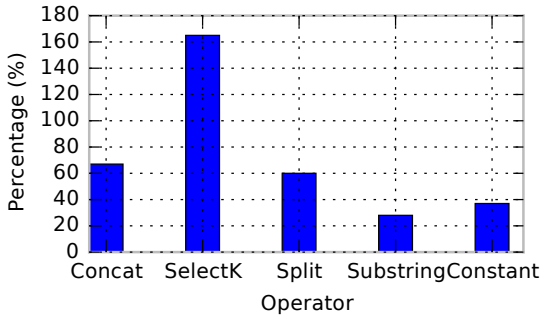


Figure 10: Distribution of operators used in ground truth join programs for all real-world join cases benchmarked.

The 4 cases for which the human expert fails to generate ground-truth programs are worth discussing. These are verified to be complicated cases, where no single string-transformation program can consistently join a large enough fraction of rows. To illustrate, in a case called `Duck-CS-Prof`[7], the join columns of two tables are professor names, which for the same person can be 2-4 tokens with many variations. For example, names can be with or without middle initials and suffixes, such as "John A. Board" in one table and "Board Jr. John A." in the other; and similarly "Yang Jun" in one table and "Jun Yang" in the other. It is hard to write a consistent program that can simultaneously join the two pairs of values above, not to mention many other name variations that are present in the data set. For cases like this, string transformation programs alone are unlikely to produce good equi-joins. The proposed auto-join could nevertheless solve these difficult cases, by generating a program to equi-join a fraction of rows, and using constrained fuzzy join to connect remaining rows, as described in Section 5.

To sum up, while it is hard to reason about "optimality" of languages, by adopting operators from well-established languages like C# and Java we have inherited desirable properties. Our analysis using benchmark data sets and human evaluator suggests that the current language is both natural and expressive for auto-join scenarios.

E. DETAILED PSEUDO CODE

In this section we present the detailed pseudo code for the Auto-Join algorithm.

E.1 Find Joinable Row Pairs

Algorithm 3 shows detailed pseudo code for finding joinable row pairs as described in Section 3.1. The code is more complex in order to cover the details that are omitted in the main paper.

KeyColumns(T) returns all the single columns that is part of a key column in the table T. Suffixes(v) returns all suffixes of a value v. QueryIndex(C, g) uses a suffix array index built for the column C, and returns a list of rows containing g. The suffix array index can cache query results to efficiently serve queries that have been seen queried.

[7]The benchmark data set from web tables is published online: https://www.microsoft.com/en-us/research/publication/auto-join-joining-tables-leveraging-transformations/

Algorithm 3 Complete pseudo code for joinable row pair

```
1: function FindJoinableRowPairs(T_s, T_t)
2:     M ← {}                                   ▷ q-gram matches
3:     for all C_s ∈ T_s do
4:         for all C_t ∈ KeyColumns(T_t) do
5:             for all v ∈ C_s do
6:                 m ← {}
7:                 for all u ∈ Suffixes(v) do
8:                     q* ← BinarySearchQ(u, C_t)
9:                     if q* < 3 or q* > Length(u) then
10:                        continue
11:                    g_u^{q*} ← u[1 : q*]
12:                    r_s ← QueryIndex(C_s, g_u^{q*})
13:                    r_t ← QueryIndex(C_t, g_u^{q*})
14:                    for all (R_s, R_t) ∈ Pairs(r_s, r_t) do
15:                        m ← ∪{(g_u^{q*}, R_s, R_t, 1/(|r_s||r_t|))}
16:                (g*, R_s*, R_t*, score) ··· ← MaxByScore(m)
17:                M ← ∪{(g*, (R_s*, R_t*), (C_s, C_t), score) ...}
18:    return SortByScore(M), GroupBy((C_s, C_t))
19: function BinarySearchQ(u, C_t)
20:    a ← 3, b ← Length(u) + 1
21:    while a < b do
22:        h ← a + (b − a)/2
23:        r_t ← QueryIndex(C_t, u[1 : h])
24:        if |r_t| > 0 then
25:            a ← h + 1
26:        else
27:            b ← h
28:    return a − 1          ▷ a is the smallest q for |r_t| = 0
```

MaxByScore returns the optimal row pairs with the highest score. Note that more than one row pairs may have the same highest score.

In Section 3.1.3, we state that the optimal prefix $g_u^{q^*}$ for a suffix u is the one that has the highest $\frac{1}{nm}$ score. Here we present our algorithm for finding $g_u^{q^*}$. First, due to the monotonicity property given by Proposition 2, the value q^* that leads to the highest score should results in the smallest possible non-zero number of matches in C_t. That is, let r_t be the matching rows in C_t returned by QueryIndex, calling QueryIndex($C_t, g_u^{q^*}$) results in $|r_t| \geq 1$, while calling QueryIndex($C_t, g_u^{q^*+1}$) results in $|r_t| = 0$. The source column C_s is not needed in finding $g_u^{q^*}$. This is because (1) there is always a match in the source column (i.e., the current row itself) so $n = |r_s| > 0$ always, and (2) when $m = |r_t| = 0$, the score $\frac{1}{nm}$ becomes undefined and thus the corresponding q is infeasible. Therefore, q^* is only obtained at the conditions mentioned above, and is not dependent on C_s. BinarySearchQ(u, C) performs the binary search of q^* that finds $g_u^{q^*}$ in C.

In practice, we found that when $q < 3$, the number of q-gram matches can be very large, severely impacting performance. Thus, we force q and the minimum length of suffixes to be at least 3.

E.2 Learn Transformations: Overall Steps

Algorithm 4 provides an overview for the transformation learning step described in Section 3.2. The LearnTransform(M) takes the q-gram matches M discovered in the previous step (Algorithm 3), which is grouped by the com-

Algorithm 4 Overall steps for transformation learning

Require: k ▷ Num. of row pairs to use in each group
Require: r ▷ Num. of example sets from each group
Require: L ▷ Max. num. of example sets to gen.
Require: b ▷ Size of an example set

function GenExampleSets(M)
  $sets \leftarrow \{\}$ ▷ Example sets
  for all $(C_s, C_j), m \in M$ **do** ▷ Grouped by col. pair
    $m \leftarrow$ TopK(m, k)
    for all $set \in$ SubsetsOfSize(m, b, r) **do**
      $sets \leftarrow sets \cup \{(set, C_t)\}$
      if $|sets| = L$ **then**
        return $sets$ ▷ Reach max. num of sets
  return $sets$
function LearnTransform(M)
  $p \leftarrow \{\}$ ▷ Learned transformations
  for all $set, C_t \in$ GenExampleSets(M) **do**
    $t \leftarrow$ TryLearnTransform(set)
    if $t =$ **null then**
      continue ▷ Failed to learn
    $C \leftarrow$ ApplyTransform(t, T_s)
    $p \leftarrow p \cup \{(C, C_t[y], t, |C \cap C_t|)\}$
  return MaxByCoverage(p) ▷ The best transform

lumn pairs. GenExampleSets generates multiple sets of examples using the groups of q-gram matches. In descending order of the average q-gram score, it goes through each group and generates a limited number of example sets. TopK takes at most k highest scored q-gram matches from each group. SubsetsOfSize creates r number of unique random subsets each with b number of q-gram matches. TryLearnTransform implements the learning algorithm, which is further expanded in Algorithm 5. ApplyTransform takes the transformation and applies it to the source table T_s. MaxBySCoverage returns the transformation (and its output) that results in the highest number of joined rows in the target table.

E.3 Learning Transformations

Algorithm 5 gives details of the TryLearnTransform procedure used in Algorithm 4, which takes a k example rows $R = \{I^i, O^i | i \in [k]\}$ as input, and produce a best-fit, low-complexity transformation as output.

The procedure is recursive and works as follows. For the given set of input/output examples, it finds the best logical operator θ that produces the most progress towards the required output strings (which in this case is some key column of the output rows). We execute the operator θ and extract partial output produced from the target output. We get what remains to the left and right in the target output, denoted as O^i_l and O^i_r, respectively. This produces two new instances of the problem with $\{I^i, O^i_l | i \in [k]\}$ and $\{I^i, O^i_r | i \in [k]\}$, which have the same structure as the original $\{I^i, O^i | i \in [k]\}$. So we recurse and invoke TryLearnTransform on the two smaller problems. The resulting operators, θ_l and θ_r, if learned successfully, are added as the left child and right child of θ, until all remaining target output have been consumed. If at certain level in the hierarchy TryLearnTransform fails to find a consistent transformation, we backtrack and proceed to find the next best logical operator.

Algorithm 5 Transformation learning by example

Require: $R = \{I^i, O^i | i \in [k]\}$ ▷ Input/output row pairs
function TryLearnTransform($R = \{I^i, O^i | i \in [k]\}$)
  while true **do**
    $\theta \leftarrow$ FindNextBestLogicalOp(R)
    $P^i \leftarrow$ ExecuteOperator(θ, I^i, O^i), $\forall i \in [k]$
    $O^i_l =$ LeftRemainder(O^i, P^i), $\forall i \in [k]$
    $\theta_l =$ TryLearnTransform($\{I^i, O^i_l | i \in [k]\}$)
    if $\theta_l =$ **null then**
      continue
    $O^i_r =$ RightRemainder(O^i, P^i), $\forall i \in [k]$
    $\theta_r =$ TryLearnTransform($\{I^i, O^i_r | i \in [k]\}$)
    if $\theta_r =$ **null then**
      continue
    θ.left_child $= \theta_l$
    θ.right_child $= \theta_r$
    return θ ▷ current root node

E.4 Fuzzy Join Optimization

Algorithm 6 provides the complete pseudo code for performing distance threshold optimization as described in Section 5. OptimizeThreshold takes the derived source column from transformation C and the target key column K, and uses binary search to find the optimal distance threshold. CheckConstraint is called at each iteration of the search to verify if the current threshold satisfies the join constraints in Equation 10.

F. LOGICAL OPERATORS

Here we give a specification of logical operators described in Section 3.2 using physical operators. Recall that the follow logical operators are used.

$\Theta = \{$Constant, Substr, SplitSubstr, SplitSplitSubstr$\}$

```
string SplitSubstr(string[] array, int k, string sep,
int m, int start, int length, Casing c) :=
Substring(SelectK(Split(SelectK(array, k), sep), m),
start, length, c)

string SplitSplitSubstr(string[] array, int k1, string
sep1, int k2, string sep2, int m, int start, int length,
Casing c) :=
Substring(SelectK(Split(SelectK(Split(SelectK(array,
k1), sep1), k2), sep2), m), start, length, c)

string Constant(string input) := input

string Substring(string[] array, int m, int start,
int length, Casing c) :=
Substring(array, m, start, length, c)
```

Note that Concat is a physical operator but not defined as a logical operator above. When programs are generated by the learning procedure (Appendix E.3), Concat is used implicitly to compose multiple logical operators in transformation programs through concatenation.

G. OPTIMIZATION FOR LEARNING

As discussed in Section 3.2, in learning transformation we conceptually enumerate all possible parameters that can be used for each logical operator. In practice physical optimizations are performed to only test meaningful parameters for a given input/output example pair.

Algorithm 6 Find the distance threshold that produces the maximum fuzzy-join coverage on the target table (Equation 9) while satisfying the join constraints (Equation 10).

Require: d_t ▷ Distance function using tokenization t
Require: $\delta \in (0.0, 1.0)$ ▷ Stopping condition
function OPTIMIZETHRESHOLD(C, K)
  $coverage_{best} \leftarrow 0$
  $a \leftarrow 0.0$, $b \leftarrow 1.0$
  $t_0 \leftarrow 0.0, t \leftarrow a + (b-a)/2$
  while $|s - s_0| > \delta$ **do**
    if CHECKCONSTRAINT(C, K, s) **then**
      $coverage \leftarrow F_{t,d,s}(C, K)$
      if $coverage > coverage_{best}$ **then**
        $coverage_{best} \leftarrow coverage$
      $b \leftarrow s$
    else
      $a \leftarrow s$
    $s_0 \leftarrow s$
    $s \leftarrow a + (b-i)/2$
  if $coverage_{best} > 0$ **then**
    return s_0 ▷ Found optimal threshold
  return -1.0 ▷ No feasible threshold found
function CHECKCONSTRAINT(C, K, s)
  $matched \leftarrow \{\}$
  for all $u \in$ DISTINCT(C) **do**
    $n' \leftarrow 0$
    for all $v \in K$ **do**
      if $d_t(u, v) \leq s$ **then**
        if $v \in matched$ **then**
          return false
        $matched \leftarrow matched \cup \{v\}$
        $n' \leftarrow n' + 1$
        if $n' > 1$ **then**
          return false
  return true

For example, for operators involving SPLIT (i.e., SPLITSUBSTR and SPLITSPLITSUBSTR), for the separator parameter, we only use punctuations and substrings that actually exist in the input string up to a certain length limit. If the input string in any recursive call does not share characters with the desired output and no CONSTANT transformation can cover all examples, no more learning needs to be performed and we backtrack for alternative transformations.

For operators involving SUBSTRING, casing parameters that are not compatible with the casing used by the output then such parameters will not need to be tested (e.g., output string is all lower case, then there is no need to test upper casing or title casing). And similar to SPLIT, when the input shares no common character with the desired output and no CONSTANT transformation can explain all examples, we also terminate and backtrack.

When attempting to generate the best logical operators by enumerating each operator, we remember the best progress that can be made using a logical operator seen so far. For each additional logical operator, we can inspect the input/output strings and derive upper bounds of how much progress can be made with that logical operator, and can early terminate testing a new logical operator if the best possible progress for it is already smaller than the current best.

H. SUCCESS GUARANTEE OF LEARNING

If the transformation required is indeed expressible in the language described before (i.e., using operators in Θ), and examples selected for learning are independent of each other, then the learning process has a high probability of success.

To show that this is the case, we start by analyzing a simplified scenario with only two operators, SUBSTR and CONCAT. Given an input string of length S_I, and an output string of length S_O that are selected as an example pair. Since only SUBSTR and CONCAT are used in generating the transformation, the desired transformation in this language is bound to be a concatenation of m substrings, where m is the size of the transformation (or the number of operators).

Our algorithm greedily selects the best operator in each step, which is the one with the most gain towards the target output. Let $\{S_1, S_2, ...S_m\}$ be the sequence of substrings generated, whose concatenation produces S_O. We now compute the probability that one substring segment S_i is correctly generated. In each step i, the reason that it may fail is because there exists another q-gram in S_I with a different match in S_O, whose length is at least $|S_i|$. The probability that this event happens for one example can be bounded by $|S_I||S_O|\frac{1}{|\Sigma|}^{|S_i|}$ [8]. For k number of independent examples, denote by $P_k(str)$ the probability of confusing a true SUBSTR operator with a different SUBSTR, $P_k(str) = |S_I|(|S_O|\frac{1}{|\Sigma|}^{|S_i|})^k$, which decreases exponentially in k. Note that $|S_I|$ is not raised to the power of k because all k examples are required to have the substring at the same position, but for S_O there is no restriction on starting positions. As a result the success probability for the i-th step is $1 - |S_I|(|S_O|\frac{1}{|\Sigma|}^{|S_i|})^k$, and the success probability for a given set of k examples across m steps is bounded by $\prod_{i\in[m]}\left(1 - |S_I|(|S_O|\frac{1}{|\Sigma|}^{|S_i|})^k\right)$. Given that we try a fixed T number of random example-sets (e.g. $T = 128$), and the overall transformation succeeds as long as one such example set can produce the correct transformation, so the overall success probability is

$$1 - \left(1 - \prod_{i\in[m]}(1 - P_k(str))\right)^T$$

We now consider the scenario with operator CONST in addition to SUBSTR and CONCAT. At step i, we may produce an incorrect CONST operator, in place of a correct SUBSTR, or a correct but different CONST. Let $P_k(const)$ be the probability. $P_k(const) = |S_O|(\frac{1}{|\Sigma|}^{|S_i|})^k$, because it also fails when there exists a constant q-gram in S_O whose length is at least $|S_i|$.

Note that $P_k(str)$ now indicates the probability of confusing a SUBSTR with CONST and a different SUBSTR, which is the same as the value computed above, because the only case where it is produced incorrectly is when the substring length exceeds that of the correct program $|S_i|$.

[8] Characters are assumed to be sampled uniformly at random from Σ for simplicity of this analysis. A different character-level distribution model can be plugged here in place of $\frac{1}{|\Sigma|}^{|S_i|}$ to produce results with the same structure and similar overall results.

$$1 - \left(1 - \prod_{i \in [m]} (1 - P_k(str) - P_k(const))\right)^T$$

We now describe the scenario with the addition of operators that use SPLIT (SPLITSUBSTR and SPLITSPLITSUBSTR). SPLIT can be viewed as modifying the relative positions of each input string, but does not alter local q-grams in other ways. So in the best case it can shift the starting position of q-grams. The probability of mistakenly producing a SPLIT instead of others is thus $P_k(split) = (|S_I||S_O|\frac{1}{|\Sigma|}^{|S_i|})^k$. Note that compared to $P_k(str)$ here $|S_I|$ is raised to the power of k, because starting position of input string is no longer constrained to be the same.

Combining, the success probability is lower bounded by

$$1 - \left(1 - \prod_{i \in [m]} (1 - P_k(str) - P_k(const) - P_k(split))\right)^T$$

which can be rewritten as

$$1 - \left(1 - \prod_{i \in [m]} \left(1 - \left(1 + (|S_I| + |S_I|^k)|S_O|^k\right) \frac{1}{|\Sigma|}^{k|S_i|}\right)\right)^T$$

The failure probability becomes exponentially small with more number of attempts T. The failure probability generally decreases when using more number of examples k, but increases with m that indicates more complex programs.

For illustration, we plug in numbers for concreteness. The program generated in Example 7 for tables in Figure 1 has the following parameters: $|S_I| = 29$, $|S_O| = 19 + 4 = 17$ (we pick the longest input/output, in this case the fourth row to bound the probability). Value of $m = 3$ since the desired program has 3 logical operators, and $|S_1| = 12, |S_2| = 4, |S_3| = 1$. Assume we use 3 examples to generate programs so $k = 3$, and $|\Sigma| = 52$. Even with $T = 1$ the success probability is ≈ 0.999 [9]. When failure probability of individual trials is more significant, with T repeated independent trials we can quickly reduce the failure probability at exponential rate, and thus produce a high overall success rate.

[9] internally when multiple operators can produce the same output sequence with the same score, CONST will be picked over other operators for the simplicity of its explanation. Thus the last operator does not have a confusion probability and will succeed with probability of 1.