APPENDIX

A. Preliminaries: Table Operators

Suppose we have two tables, T_1, T_2, that share common columns C, and are in their minimal forms in which there are no duplicates and no tuples that can be subsumed or complemented. We show that for each pairwise table operator, Inner Union, Inner Join, Left Join, Outer Join, Cross Product, there exists an equivalent query consisting of Outer Union and/or unary operators. (SP of SPJU queries are accounted for by the unary operators).

Lemma 11 (Inner Union): Inner Union($\cup$): it is known that if the schemas of two tables are equal, then Inner Union = Outer Union

Lemma 12 (Inner Join): Inner Join ($\bowtie$):

$$T_1 \bowtie T_2 = \sigma(T_1.C = T_2.C \neq \perp, \beta(\kappa(T_1 \uplus T_2))) \quad (7)$$

Lemma 13 (Left Join): Left Join ($\rtimes$) [32]:

$$T_1 \rtimes T_2 = \beta((T_1 \bowtie T_2) \uplus T_1) \quad (8)$$

Lemma 14 (Outer Join): Full Outer Join ($\rtimes\ltimes$) [32]:

$$T_1 \rtimes\ltimes T_2 = \beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2) \quad (9)$$

Lemma 15 (Cross Product): Cross Product($\times$): We denote columns in T_1 and T_2 as $T_1.C$ and $T_2.C$, respectively. Consider a constant column c.

$$T_1 \times T_2 = \kappa(\pi((T_1.C, c), T_1) \uplus \pi((T_2.C, c), T_2)) \quad (10)$$

Thus, $\uplus, \sigma, \pi, \kappa, \beta$ operators form queries that are equivalent to all SPJU queries.

1) Proof of Lemma 12 [Inner Join]: Given two tables T_1, T_2 that join on a set of common columns C, such that T_1, T_2 are in their minimal forms in which they contain no duplicate tuples and no tuples can be subsumed or complemented , $T_1 \bowtie T_2$ can be expressed by an equivalent query containing Outer Union, complementation, and subsumption. Specifically, $T_1 \bowtie T_2$ is equivalent to query $\sigma(T_1.C = T_2.C \neq \perp, \beta(\kappa(T_1 \uplus T_2)))$.

Proof: We first prove that all tuples in $T_1 \bowtie T_2$ are contained in $\sigma(T_1.C = T_2.C \neq \perp, \beta(\kappa(T_1 \uplus T_2)))$. Let tuple $t \in T_1 \bowtie T_2$, such that join columns C's values in t appear in both $T_1.C$ and $T_2.C$, and are non-null: $t.C \in T_1.C \cap T_2.C$ s.t. $t.C \neq \perp$.

When applying $\beta(\kappa(T_1 \uplus T_2))$, only tuples with common non-null values $T_1.C_i = T_2.C_i \neq \perp$ in same column(s) i are complemented and subsumed. This is similar to tuple t, which is formed by joining on $T_1.C_i = T_2.C_i$. Thus, tuple t is derived by selecting on tuples from $\beta(\kappa(T_1 \uplus T_2))$ with non-null C values in both $T_1.C$ and $T_2.C$, so $t \in \sigma(T_1.C = T_2.C \neq \perp, \beta(\kappa(T_1 \uplus T_2)))$.

Next, we show that all tuples in $\sigma(T_1.C = T_2.C \neq \perp, \beta(\kappa(T_1 \uplus T_2)))$ are found in $T_1 \bowtie T_2$. Let tuple $t' \in \sigma(T_1.C = T_2.C \neq \perp, \beta(\kappa(T_1 \uplus T_2)))$. Here, all C values in t' are non-null values found in both $T_1.C$ and $T_2.C$ as a result of selection. From $\beta(\kappa(T_1 \uplus T_2))$, t' contains all values from all columns in T_1 and T_2 in a single tuple, formed by complementing and subsuming based on common C values. Thus, $t' \in T_1 \bowtie T_2$.

We have thus shown that all tuples from $T_1 \bowtie T_2$ are found in $\sigma(T_1.C = T_2.C \neq \perp, \beta(\kappa(T_1 \uplus T_2)))$ and vice versa, and so $T_1 \bowtie T_2$ is an equivalent query to $\sigma(T_1.C = T_2.C \neq \perp, \beta(\kappa(T_1 \uplus T_2)))$. ■

2) Proof of Lemma 13 [Left Join]: Given two tables T_1, T_2 that join on a set of common columns C, such that T_1, T_2 are in their minimal forms in which there are no duplicates and no tuples can be subsumed or complemented , $T_1 \rtimes T_2$ can be expressed by an equivalent query containing Outer Union and subsumption. Specifically, $T_1 \rtimes T_2$ is equivalent to query $\beta((T_1 \bowtie T_2) \uplus T_1)$.

Proof: We first prove that the resulting table of $T_1 \rtimes T_2$ is contained in the resulting table of $\beta((T_1 \bowtie T_2) \uplus T_1)$:

Let tuple $t \in T_1 \rtimes T_2$. There are two cases for join column C's values in tuple t: $t.C \in T_1.C \cap T_2.C$ (i.e., $t.C$ values are in both $T_1.C$ and in $T_2.C$) and $t.C \in T_1.C \setminus T_2.C$ (i.e., $t.C$ values are only in $T_1.C$ and not in $T_2.C$). Since we are performing left join on T_1 and T_2, $t.C \notin T_2.C \setminus T_1.C$.

1) $t.C \in T_1.C \cap T_2.C \implies t \in (T_1 \bowtie T_2)$. Since t is in the inner join result and contains more non-Null values than other tuples with C values only in T_1 or T_2, it would not be subsumed when applying $\beta((T_1 \bowtie T_2) \uplus T_1)$.
2) $t.C \in (T_1.C \setminus T_2.C) \implies t \in \beta((T_1 \bowtie T_2) \uplus T_1)$. Since T_1 is in its minimal form, and t does not share any C values with any tuple in T_2, it is not subsumed when applying β to $(T_1 \bowtie T_2) \uplus T_2$, and thus appear as is in $\beta((T_1 \bowtie T_2) \uplus T_1)$.

Thus, all tuples from $T_1 \rtimes T_2$ are contained in the resulting table of $\beta((T_1 \bowtie T_2) \uplus T_1)$.

Next, we show that the resulting tuples of $\beta((T_1 \bowtie T_2) \uplus T_1)$ are contained in the resulting table of $T_1 \rtimes T_2$.

Let's consider tuple $t' \in \beta((T_1 \bowtie T_2) \uplus T_1)$. There are two cases for C values in tuple t': $t'.C \in T_1.C \cap T_2.C$ and $t'.C \notin T_1.C \cap T_2.C$.

1) $t'.C \in (T_1.C \cap T_2.C) \implies t' \in (T_1 \bowtie T_2)$. Since $(T_1 \bowtie T_2) \subseteq (T_1 \rtimes T_2)$, $t' \in (T_1 \rtimes T_2)$.
2) All tuples in $((T_1 \bowtie T_2) \uplus T_1)$ are either subsumed by tuples from $(T_1 \bowtie T_2)$, or are in $T_1 \setminus (T_1 \bowtie T_2)$. Thus, $t'.C \notin T_1.C \cap T_2.C \implies t' \in T_1 \setminus (T_1 \bowtie T_2) \implies t' \in T_1 \rtimes T_2$.

Thus, all tuples from $\beta((T_1 \bowtie T_2) \uplus T_1)$ are contained in the resulting table of $T_1 \rtimes T_2$.

Now that we have shown that tuples from $T_1 \rtimes T_2$ are contained in the resulting table of $\beta((T_1 \bowtie T_2) \uplus T_1)$ and vice versa, we have shown that $\beta((T_1 \bowtie T_2) \uplus T_1)$ is an equivalent query to $T_1 \rtimes T_2$. ■

3) Proof of Lemma 14 [Outer Join]: Given two tables T_1, T_2 that join on a set of common columns C, such that T_1, T_2 are in their minimal forms in which there are no duplicates and no tuples can be subsumed or complemented, $T_1 \rtimes\ltimes T_2$ can be expressed by an equivalent query containing Outer Union and subsumption. Specifically, $T_1 \rtimes\ltimes T_2$ is equivalent to query $\beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$.

Proof: We first prove that the resulting table of $T_1 \mathbin{⟗} T_2$ is contained in the resulting table of $\beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$:

Let tuple $t \in T_1 \mathbin{⟗} T_2$. There are three cases for join column C's values in tuple t: $t.C \in T_1.C \cap T_2.C$ (i.e., $t.C$ values are in both $T_1.C$ and in $T_2.C$), $t.C \in T_1.C \setminus T_2.C$ (i.e., $t.C$ values are only in $T_1.C$ and not in $T_2.C$), and $t.C \in T_2.C \setminus T_1.C$ (i.e., $t.C$ values are only in $T_2.C$ and not in $T_1.C$).

1) $t.C \in T_1.C \cap T_2.C \implies t \in T_1 \bowtie T_2$. Tuple t is a result of inner joining two tuples from T_1, T_2 on shared values in common columns C. This is similar to taking $T_1 \uplus T_2$, and applying subsumption and complementation on tuples with shared values in C (Lemma 12) to get t. Since t does not share any values in C with other tuples, it cannot be subsumed. Thus, $t \in \beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$.
2) $t.C \in T_1.C \setminus T_2.C \implies t \in (T_1 \mathbin{⟗} T_2) \setminus (T_1 \bowtie T_2)$. When we take $(T_1 \bowtie T_1) \uplus T_1$, we append all tuples from T_1 to $T_1 \bowtie T_2$. After applying subsumption, all tuples from T_1 that are used in $T_1 \bowtie T_2$ are subsumed by tuples from $T_1 \bowtie T_2$ on shared values in C. Thus, the only tuples remaining are tuples like t in $(T_1 \mathbin{⟗} T_2) \setminus (T_1 \bowtie T_2)$. Since t does not share any common values with any tuple in T_2, it is not subsumed when taking $\beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$, and so $t \in \beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$.
3) $t.C \in T_2.C \setminus T_1.C \implies t \in (T_2 \mathbin{⟗} T_1) \setminus (T_1 \bowtie T_2)$. Taking the subsumption of $\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2$ removes all tuples from T_2 that are subsumed by tuples in $T_1 \bowtie T_2$. Since the remaining tuples in T_2 cannot be subsumed by any tuple from T_1 not in $T_1 \bowtie T_2$, $t \in (T_2 \mathbin{⟗} T_1) \setminus (T_1 \bowtie T_2)$. Thus, $t \in \beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$.

Thus, all tuples from $T_1 \mathbin{⟗} T_2$ are contained in the resulting table of $\beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$.

Next, we show that all tuples in $\beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$ are contained in the resulting table of $T_1 \mathbin{⟗} T_2$. Let's consider tuple $t' \in \beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$. There are two cases for C values in tuple t': $t'.C \in T_1.C \cap T_2.C$ and $t'.C \notin T_1.C \cap T_2.C$.

1) $t'.C \in (T_1.C \cap T_2.C) \implies t' \in (T_1 \bowtie T_2)$. Since $(T_1 \bowtie T_2) \subseteq (T_1 \mathbin{⟗} T_2)$, $t' \in (T_1 \mathbin{⟗} T_2)$.
2) All tuples in $((T_1 \bowtie T_2) \uplus T_1) \uplus T_2$ are either subsumed by tuples from $(T_1 \bowtie T_2)$, are in $T_1 \setminus (T_1 \bowtie T_2)$, or are in $T_2 \setminus (T_1 \bowtie T_2)$. Thus, $t'.C \notin T_1.C \cap T_2.C \implies t' \in (T_1 \uplus T_2) \setminus (T_1 \bowtie T_2) \implies t' \in T_1 \mathbin{⟗} T_2$.

Thus, all tuples from $\beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$ are contained in the resulting table of $T_1 \mathbin{⟗} T_2$.

Now that we have shown that tuples from $T_1 \mathbin{⟗} T_2$ are contained in the resulting table of $\beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$ and vice versa, we have shown that $\beta(\beta((T_1 \bowtie T_2) \uplus T_1) \uplus T_2)$ is an equivalent query to $T_1 \mathbin{⟗} T_2$.

■

4) Proof of Lemma 15 [Cross Product]: Given two tables T_1, T_2, each with columns C_{T_1}, C_{T_2} respectively and do not share any columns, and a constant column c, $T_1 \times T_2$ can be expressed by an equivalent query containing Outer Union, projection, and complementation. Specifically, $T_1 \times T_2$ is equivalent to query $\kappa(\pi((C_{T_1}, c), T_1) \uplus \pi((C_{T_2}, c), T_2))$.

Proof: Since T_1 and T_2 do not share any columns, the complementation operator cannot be applied to $T_1 \uplus T_2$. Thus, we project on all columns C_{T_1} and constant column c in T_1, and columns C_{T_2}, c in T_2. This way, T_1, T_2 now share all values in c and we can apply complementation on $\pi((C_{T_1}, c), T_1) \uplus \pi((C_{T_2}, c), T_2)$ since T_1, T_2. Thus, we iteratively apply complementation on all tuples from T_1 on all tuples from T_2 to form all tuples in $T_1 \times T_2$. Recall that in every tuple in $T_1 \times T_2$, every value in $t.C_{T_1}$ is from T_1 and every value in $t.C_{T_2}$ is from T_2. Therefore, every tuple in $T_1 \times T_2$ is contained in $\kappa(\pi((C_{T_1}, c), T_1) \uplus \pi((C_{T_2}, c), T_2))$ and every tuple in $\kappa(\pi((C_{T_1}, c), T_1) \uplus \pi((C_{T_2}, c), T_2))$ is contained in $T_1 \times T_2$, and so $\kappa(\pi((C_{T_1}, c), T_1) \uplus \pi((C_{T_2}, c), T_2))$ is an equivalent query to $T_1 \times T_2$. ■

B. Set Similarity

Algorithm 3: Set Similarity

```
1  Input: 𝒯 = {T_1, T_2, ... T_n}: set of data lake tables; S:
     Source Table; τ: Similarity Threshold
2  Output: 𝒯' = {T_1, T_2, ... T_n}: a set of candidate tables
     with high syntactic overlap with S
3  𝒯'_scores ← {} //Store a list of scores for each candidate table
4  for all S columns c ∈ C do
5  |   𝒯_C, overlapScores ← SetOverlap(𝒯, c, τ)
6  |   𝒯_C, diverseOverlapScores
   |     ← diversifyCandidates(𝒯_C, c, τ)
7  |   for all tables T ∈ 𝒯_C do
8  |   |   𝒯'_scores[T] += diverseOverlapScores[T]
9  Order 𝒯'_scores by average diverseOverlapScores, in descending
     order
10 𝒯' ← keys(𝒯'_scores)
11 for all tables T ∈ 𝒯' do
12 |   alignedTuples ← tuples in T that contain S's column
   |     values
13 |   if set overlap of T values in alignedTuples with S < τ
   |     then
14 |   |   Discard T;
15 |   Remove T if its values are contained in another table
   |     T' ∈ 𝒯'
16 |   Rename T columns to aligned S columns
17 return 𝒯';
```

We find candidate tables with values that have high set overlap with those in a Source Table. As shown in Algorithm 3, we perform Set Similarity with an input set of data lake tables $\mathcal{T}$, the Source Table S, and a similarity threshold τ (Line 1), and output a set of candidate tables (Line 2). We first find a set of candidate tables, where each table contains a column whose set overlap with a column from S (overlapScore) is above a specified threshold (Lines 4-8). This can be done efficiently with a system like JOSIE [10] that computes exact set containment or MATE [44] that supports multi-attribute joins. In addition, when finding tables with columns that have a high set overlap with columns in S, we call diversifyCandidates() (Line 6) to ensure that each candidate table not only has a high overlap with S, but also has minimal overlap with the previous candidates, shown in Diversify Candidates Algorithm 4.

Algorithm 4: Diversify Candidates

Input: c: column from Source Table; $\mathcal{T}_C = \{T_1, T_2, \dots T_n\}$: set of candidate tables with columns having high overlap with c; τ: Similarity Threshold
Output: $\mathcal{T}'_C = \{T_1, T_2, \dots T_n\}$: a set of diverse candidate tables
$\mathcal{T}_{\text{scores}} \leftarrow \{\}$
for *all tables* $T \in \mathcal{T}_C$ **do**
    $C \leftarrow$ column from T with highest set overlap with c
    $\text{Ind}_T \leftarrow$ index of T in $\mathcal{T}_C$
    if $Ind_T = 0$ **then**
        Continue;
    $C_{\text{prev}} \leftarrow$ column from $\mathcal{T}_C[\text{Ind}_T - 1]$ with highest set overlap with c//Get column from previous candidate table with high overlap with c
    prevColOverlap $\leftarrow (C \cap C_{\text{prev}})/|C|$ //Set overlap with previous column
    sourceColOverlap $\leftarrow (C \cap c)/|c|$ //Set overlap with column from Source table
    if $sourceColOverlap < \tau$ **then**
        Continue;
    overlapScore $\leftarrow$ sourceColOverlap – prevColOverlap
    $\mathcal{T}_{\text{scores}}[T] \leftarrow$ overlapScore
Order $\mathcal{T}_{\text{scores}}$ by values in descending order
$\mathcal{T}'_C \leftarrow \mathcal{T}_{\text{scores}}$.keys
return $\mathcal{T}'_C$;

After we find candidate tables for each column in the Source Table, we average over all overlap scores such that each is for a Source Table's column with which they share many values, and rank them in descending order of averaged scores (Line 9). With a set of candidate tables, we find tuples in each candidate table that contain column values from S. Within these aligned tuples, we check if each aligned column in a candidate table, with respect to a column in S, still has high set overlap (above threshold τ). If not, we remove them (Line 14). Next, we remove any subsumed candidate table, whose columns and column values are all contained in another candidate table (Line 15). We then rename each candidate tables' columns to the names of S's columns with which they align (Line 16), thus implicitly performing schema matching between S's columns and the columns from the candidate tables that have overlapping values with S's columns. Finally, we return the set of candidate tables.

C. Expanding Candidate Tables

In order to represent candidate tables as matrices, their tuples need to align with those in the Source Table. However, not all candidate tables may share a key column with the Source Table. Thus, we need to join a given candidate table that does not share a key column with the Source Table with those that do. This way, tuples from all candidate tables can be aligned with tuples from a Source Table using key values.

As illustrated in Expand Algorithm 5, we traverse a graph that consists of candidate tables as nodes and we find a join path between candidate tables that do not have a Source Table's key (start nodes), and candidate tables that do (potential end nodes). If two candidate tables can join on common columns, their nodes are connected with an edge. For each edge, we use standard join-size cardinality estimation to find edge weights [75].

After we find a path from a start node to an end node, we iteratively join all tables in the path, resulting in a table that shares a key column with the Source Table. This way, all candidate tables share a key column with the Source Table.

Algorithm 5: Expand

Input: $\mathcal{T} = \{T_1, T_2, \dots T_n\}$: set of candidate tables; k: Source Table's key column(s)
Output: $\mathcal{T}_k = \{T_1, T_2, \dots T_n\}$: set of candidate tables that all now contain Source Table's key column
//Initialize Graph
nodes = candidate tables
edges = tables that have joinable columns
edge Weights = value overlap of joinable columns
$start\_nodes$ = {candidate tables that do not contain k}
end_nodes = {candidate tables that contain k}
for *each* $start$ *in* $start\_nodes$ **do**
    //Initialize sets and dictionaries
    $visited \leftarrow$ set() //visited nodes
    $node\_weights \leftarrow \{\}$ //maximum weights before each node
    $descendant \leftarrow \{\}$ //best child for each node
    $max\_weight \leftarrow 0$ //Maximum weight found so far
    $end\_node \leftarrow$ None //end node for a given start node
    //Initialize $stack$ for DFS
    $stack \leftarrow stack + start$
    $visited \leftarrow visited + start$
    while $stack$ *is not empty* **do**
        $node \leftarrow stack.pop()$
        $unvisited\_children \leftarrow$ children of $node$ not in $visited$
        //Current child's weight is the weight of the path so far, including the edge weight between node and current child
        for *each* $child$ *in* $unvisited\_children$ **do**
            $child\_weight \leftarrow node\_weights[node] + weight(node, child)$
            //update descendant if it contains the maximum sum of weights so far
            if $child\_weight > node\_weights[child]$ **then**
                $node\_weights[child] \leftarrow child\_weight$
                $descendant[child] \leftarrow node$
                if $child$ *is in* end_nodes **then**
                    if $child\_weight > max\_weight$ **then**
                        //child has k and the maximum weighted path so far
                        $max\_weight \leftarrow child\_weight$
                        $end\_node \leftarrow child$
                $stack \leftarrow stack + child$
            $visited \leftarrow visited + child$
    if end_node *is not null* **then**
        //reconstruct path with maximum sum of weights by reversing path, starting with found end node
        $path \leftarrow []$
        $current\_node \leftarrow end\_node$
        while $current\_node$ *is in* $descendant$ **do**
            $path \leftarrow path + current\_node$
            $current\_node \leftarrow descendant[current\_node]$
        $path.reverse()$
        $table \leftarrow$ first node in $path$
        for *each* $join\_table$ *in* $path[1:]$ **do**
            $table \leftarrow join(table, join\_table)$
        $\mathcal{T}_k \leftarrow \mathcal{T}_k + table$

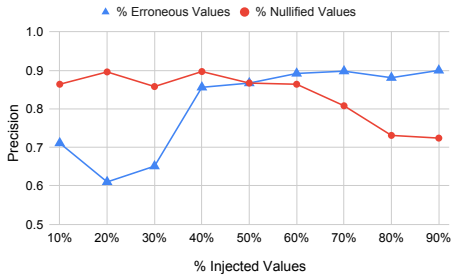


Fig. 8: Gen-T Precision as TP-TR Med has different % of Erroneous (blue triangles) and Nullified Values (red circles).

D. Two-Valued vs. Three-Valued Matrix Representations

After aligning tuples in candidate tables to Source Table's tuples that share the same key values, we can represent candidate tables as matrices to show how similar the values in candidate tables are to those in the Source Table. These matrices have the same dimensions and indices as the Source Table. First, we consider matrices that consist of binary values, where a 0 in tuple i, column j represents a value at index (i, j) in the candidate table that is different from the value in the same position in the Source Table, and a 1 represent common values in the same indices.

However, populating alignment matrices with binary values does not fully encode how many values in candidate tables *differ* from those in the Source Table. Specifically, this representation does not distinguish between nullified values in candidate tables (null values in candidate tables at index (i, j) where there is a non-null value in the Source Table at the same position), and erroneous values (different non-null values in candidate tables at index (i, j) from those in Source Tables at the same position).

Instead, we encode matrix representations using three values, where at a given index in an aligned tuple, there is a 1 for a value shared between a candidate table and the Source Table, 0 for a null value in the candidate table where there is a non-null value in the Source Table, and -1 for a non-null value in the candidate table that differs from the value in the Source Table.

E. Metrics

Conditional KL-divergence: Given column C shared between a Source Table and a reclaimed table T, suppose we have probability distributions, $\mathcal{P}$ for C in the Source Table and $\mathcal{Q}$ for C in the reclaimed table. We condition on the key values in key column K. The conditional KL-divergence (or conditional relative entropy) between $\mathcal{P}$ and $\mathcal{Q}$ of sample space X of column C conditioned on key K is as follows:

$$D_{KL}(Q||P) = -\sum_{x \in X, k \in K} P(x|k) \log\left(\frac{Q(x|k)(1 - Q(\neg x|k)}{P(x|k)}\right) \tag{11}$$

Given n non-key columns $\mathcal{C}$ in a Source Table we take the average D_{KL} for each column divided by the probability of a key value in T matching a key value from the Source Table ($Q(K)$) and the number of non-key columns (n). Then, the conditional KL-divergence of the reclaimed table is as follows:

$$D_{KL}(T) = \frac{D_{KL}(Q_1||P_1) + D_{KL}(Q_2||P_2) + \cdots + D_{KL}(Q_n||P_n)}{Q(K) * n} \tag{12}$$

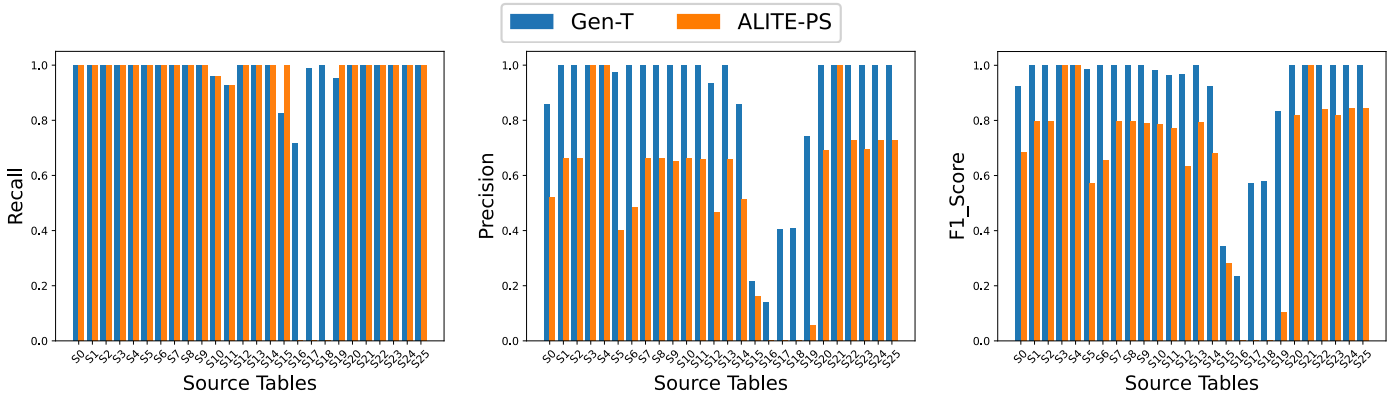
The conditional KL-divergence of the reclaimed table is a score $\in [0, \infty)$, with 0 being the ideal score. There is no upper limit on this metric since it naturally approaches ∞ when no key value from the Source Table is found in the reclaimed table.

F. Effectiveness: Ablation Study

Tuning % Erroneous vs. Nullified Values: We further analyze Gen-T's performance on data lake tables with different number of erroneous and nullified values in TP-TR Med tables (Figure 8). So far, TP-TR Med tables have 50% erroneous values in erroneous versions and 50% nulls in nullified versions (intersection point on the graph where Gen-T has 0.867 Precision). Now, we tune the percentage of values replaced with non-null, random strings (blue line in Figure 8) in erroneous versions, while the nullified versions always contain 50% nulls. Similarly, we tune the percentage of values replaced with nulls (red line in Figure 8) while holding the erroneous versions constant. For Gen-T to produce a perfect reclamation of a Source Table, it should only have originating tables with injected nulls so that these nulls can be replaced with correct values during table integration.

As data lake tables have more erroneous values (blue line), Gen-T is more likely to contain tables with nullified tuples in its set of originating tables, which results in an integrated table with higher precision. On the other hand, as we tune the percentage of values replaced with nulls (red line), precision decreases. As more nulls are injected, these tables also have fewer correct values. Gen-T is thus more inclined to have originating tables with 50% erroneous values, or 50% correct values, leading to a final integration with lower precision.

Effectiveness of Pruning in Gen-T: For a more detailed analysis, we analyze Recall, Precision, and F1-Scores of Gen-T and baseline, ALITE-PS, on TP-TR Med for each of its 26 Source Tables (Figure 9). Note that ALITE-PS directly integrates a set of candidate tables, whereas Gen-T first prunes the set of candidate tables to a set of originating tables before performing table integration. Gen-T outperforms ALITE-PS in Precision for all Source Tables, and outperforms ALITE-PS in Recall for 24 of 26 total Source Tables. This shows that ALITE-PS, which directly integrates candidate tables without pruning, reclaims more Source Tuples than Gen-T, which does include a pruning step, for only a few Source Tables. Also, Gen-T outperforms ALITE-PS in F1-Score for all Source Tables (Figure 9(c)), showing that even if ALITE-PS outperforms Gen-T in Recall, it does not impact the F1-Score.



(a) Recall **(b)** Precision **(c)** F1 Score

Fig. 9: Recall, Precision, and F1 Scores of Gen-T and ALITE-PS for each Source Table in TP-TR Med benchmark.