On Discovering Provenance of Data Warehouse Objects Created by Select-Project Queries

Witold Andrzejewski[1,*], Pawel Boinski[1] and Robert Wrembel[1]

[1]Poznan University of Technology

Abstract

Data lineage is the set of techniques that allow tracking data transformations along a data integration (DI) workflow. Despite substantial research that has already been done on this topic, discovering missing or broken links between database objects in large data repositories still remains challenging. In a data warehouse system, objects like materialized views are created from other objects by means of SQL select-project-join queries. As a result, dependencies between objects are complex, thus difficult to discover and maintain. Moreover, DI workflows often create auxiliary temporary objects (e.g., temporary tables, global variables in a database programming language) and they invoke codes (e.g., table functions, procedures with cursors) that produce volatile data. These volatile data and data structures are used to build permanent data sets and structures, but they cease to exist after terminating a given DI task. As a consequence, relationships between such objects cannot be automatically recorded. This paper provides algorithms that discover relationships between data warehouse objects. We address objects that were created by means of SQL projection-selection queries, where projection of at least one attribute is performed via a linear function. In particular we contribute: (1) an algorithm that discovers the slope and intercept of a linear function that was used in a project query and (2) an algorithm that recovers the selection condition used in a select query.

Keywords

data lineage, data provenance, broken lineage, data integration workflow, data warehouse, select-project query, linear regression

1. Introduction

Data lineage (also referred to as data provenance [1, 2]) encompasses the systematic documentation of the lifecycle of data. This documentation spans the data lifecycle from its origin in a data source, through various transformations applied within data integration (DI) workflows, to its final destination, which is typically a data warehouse (DW), data lake, or data lakehouse [3, 4, 5, 6]. Data lineage techniques are fundamental to data management, governance, and compliance, as they provide insights into data origins and processing logic. Consequently, data lineage is an indispensable instrument for validating data quality, diagnosing processing anomalies, and providing regulatory bodies (especially in the financial sector) with auditable evidence of data correctness and integrity.

While existing data lineage research primarily focuses on identifying and managing relationships between individual data records, these relationships are fundamentally determined by the underlying dependencies between database objects, such as tables, views, and materialized views. To the best of our knowledge, the systematic discovery of these structural, object-level relationships remains an underexplored area in the literature.

DI workflows often create auxiliary temporary objects (e.g., temporary tables in a staging area) and they call predefined functions or user-defined functions (UDFs) that produce volatile tabular data (e.g., by table functions). Often, these volatile data and data structures are used to build permanent data sets and structures. Typically, auxiliary temporary objects (temporary tables, table functions, procedures with cursors and global variables) cease to exist after terminating a given DI task. As a consequence, **data lineage gets broken**. To the best of our knowledge, the state-of-the-art research solutions for data

Published in the Proceedings of the Workshops of the EDBT/ICDT 2026 Joint Conference (March 24-27, 2026), Tampere, Finland

*Corresponding author

witold.andrzejewski@cs.put.poznan.pl (W. Andrzejewski); pawel.boinski@cs.put.poznan.pl (P. Boinski); robert.wrembel@cs.put.poznan.pl (R. Wrembel)

0000-0001-9486-929X (W. Andrzejewski); 0000-0003-4914-9394 (P. Boinski); 0000-0001-6037-5718 (R. Wrembel)

lineage have not addressed the problem of a *broken lineage* [7, 8]. What is more, none of the commercial or open-source DI engines allows for the discovery of the broken lineage. The broken lineage prevents users from checking data quality and from managing dependencies between database objects, which in turn increases system development costs.

This research work was motivated by a real problem of the broken lineage, faced by a company in the financial sector. The company uses a DW that integrates data from multiple external systems (mainly relational databases). New DW objects are created to meet the demand for new analytical reports. These new objects are based on existing DW and source database objects. For instance, a DW in the company IT architecture encompasses over 5,000 views and materialized views as well as over 11,000 stored procedures and functions, totaling more than 2 million lines of code. Remarkably, a DW of this size is still classified as a small DW.

The problem addressed in this paper is aggravated by the fact that institutions typically maintain multiple subject-oriented DWs. The vast number of DW objects and source objects used leads to an overwhelming complexity of dependencies between them. This creates a significant challenge in managing source-to-target relationships, as a single source object often feeds multiple targets across multiple warehouses. Furthermore, objects in one DW may be derived from multiple source objects and/or objects from other DWs.

In this paper, we introduce a **method for discovering missing** and **broken lineage links** between database objects (Section 3). We focus on a frequent real-world scenario: a new relation is created to store a specific subset (a selection) of data from a source table, but the values are transformed rather than just copied. These transformations typically involve linear unit changes, such as converting currencies, temperature scales, or time zones. For example, a database might have a `raw_trades` table recording all stock exchange activity. To help analysts, a developer might create a `large_trades_PLN` table that only includes transactions over 10,000 shares and converts prices from USD to PLN. Often, the metadata or "context" columns are dropped during this process, making it difficult to prove where the data actually came from. This creates the broken lineage problem.

The method presented in this paper is capable of solving the problem by discovering the slope and intercept of the linear function used in the projection as well as recovering the selection criteria used. In the stock market example, our approach would successfully identify the USD-to-PLN conversion rate and the 10,000-share filter used to create the new table.

The experimental evaluation of the method is reported in Section 4. The results show the algorithms performance w.r.t. to several distinct parameters and demonstrate that the selection-projection query is reconstructed correctly. We conclude the paper with a short summary in Section 5.

2. Related work

Three fundamental solutions for managing data provenance in databases have been proposed, namely: annotation-based, inversion-based, and lineage graphs. The *annotation-based* solutions annotate input data with information about their origin. Then, these annotations are propagated by a database engine through the entire data processing pipeline. Historically, first annotation-based solutions include Polygen [5] and DBNotes [9]. More recent works using the same concept include [10, 11, 12]. A common approach to annotation-based data provenance representation is the mathematical model known as *semiring* [13, 14]. The *inversion-based* solutions invert queries trying to retrospectively infer the original data that produced a given result, e.g., [15, 6]. Finally, in the *lineage graph* approach, provenance is registered during query processing and is saved in the form of a graph for further analysis [16].

[17] systematizes data lineage characteristics, such as (1) lineage type, (2) lineage granularity, (3) types of tracked data transformations, and (4) the point in time where the lineage information is stored. Paper [18] summarizes several most relevant works on data provenance.

Data science pipelines typically process data using scripting languages (e.g., Python, R) where data provenance also needs to be maintained, e.g., [19, 20, 21]. Paper [22] proposes that a user expresses

provenance information need, independent of the instrumentation and tracing of scripts.

The standardization of provenance information was initiated in the Open Provenance Model (OPM) was created [23]. OPM uses a directed acyclic graph to represent provenance information. The OPM was replaced by the W3C standard PROV [24, 25], to facilitate provenance data exchange over the Web.

Query Reverse Engineering (QRE) is a related research area. State-of-the-art QRE techniques (e.g., [26, 27, 28]) primarily focus on synthesizing queries involving complex structural operations like joins and aggregations. However, they lack the capability to discover algebraic data transformations that we address in this paper.

3. Contribution

In [7], we presented an approach to detect whether there is a possibility that a column from one table was derived from another table column via selection and linear transformation. Moreover, we provided a method for estimating slope and intercept for the linear transformation under the assumption that the slope is positive.

In this paper, we build upon the previous solution to provide the following improvements:

- significantly more accurate detection of slope and intercept w.r.t. the previous approach;
- the detection of negative slopes (the previous method allowed only for positive slopes);
- the reconstruction of selection conditions.

In Section 3.1, we establish the notation used in the subsequent discussion and formally define the problem addressed in this paper. In Section 3.2, we provide an algorithm for determining the slope and intercept. In Section 3.3, we provide the important component of this algorithm (a verification step). In Section 3.4, we present an algorithm for reconstructing the selection conditions. Finally, in Section 3.5 we discuss the limitations of the current solution and provide topics for future work.

To the best of our knowledge, the only paper directly related to this problem is [7].

3.1. Notations and problem definition

Assume two relations $S(s_1, s_2, \ldots, s_n, x)$ and $T(t_1, t_2, \ldots, t_m, y)$. For conciseness, we denote the set of all s_i attributes as S^A and the set of all t_i attributes as T^A. We say that there is a *relationship* between S and T if $|S| \geq |T|$ and $T = \Pi_{f^1(S^A) \text{ as } t_1, \ldots, f^m(S^A) \text{ as } t_m, ax+b \text{ as } y}(\sigma_c(S))$, where the functions f^i are arbitrary and unknown, and the coefficient a is non-zero. In other words, attributes t_i may, or may not, be computed from S^A in an arbitrary manner. We assume that the selection condition c is represented in disjunctive normal form, where each clause is a conjunction of literals, and each literal is an inequality of the form $s_i\{\leq, >\}v$, where v belongs to the domain of s_i.

In this paper, we present algorithms to solve the following problem: given relations S and T as defined above, find the slope a, the intercept b and the selection condition c if a relationship exists, or return an indication that T was not derived from S otherwise.

3.2. Finding slope and intercept

Basic algebra yields, that given any two pairs of x and y values, e.g., (x_1, y_1) and (x_2, y_2), one can easily derive the slope a and intercept b as:

$$a = \frac{y_1 - y_2}{x_1 - x_2} \tag{1}$$

$$b = y_1 - ax_1 = y_2 - ax_2 \tag{2}$$

Under the assumption that the relationship exists, every row in T corresponds to some row in S. Thus, in order to find the slope and intercept, it is sufficient to choose any two rows in T, and treat attribute the y attribute values from these rows as y_1 and y_2. Finding x_1 and x_2 from the relation S is more

challenging, since we do not know which rows were chosen by the selection condition c. Thus, we need to map every 2-permutation (x_1, x_2) of values in the x attribute to the y_1 and y_2 values obtained previously, compute a and b and verify whether, for each row in T, some x value in S allows us to compute y. If the verification is successful, the linear function parameters are found.

Unfortunately, the proposed solution, while accurate, is highly complex. There are $O(|S|^2)$ pairs of x_1 and x_2. Moreover, for each such pair, a verification must be performed, which requires checking if all y values in T are properly explained. Thus, the verification requires at least $|T|$ steps. In fact, the actual complexity of this verification step is even higher, as we show in Section 3.3. In order to mitigate this problem, we propose the following solutions.

First, in [7] we have proposed to use two-sample Kolmogorov-Smirnov test to detect whether there is a possibility that a relationship exists between the two tables. The complex approach should be used only if such a possibility exists.

Second, the number of checked 2-permutations can be limited using the following approach. Choose the extreme y values as y_1 and y_2, i.e.:

$$y_1 = \mathcal{G}_{min(y)}(T) \tag{3}$$

$$y_2 = \mathcal{G}_{max(y)}(T) \tag{4}$$

Since a linear function is monotonic, the corresponding x_1 and x_2 should be the closest to the extreme values in x attribute as well (though not necessarily equal to them). Assuming the slope a is positive, the correct x_1 corresponding to y_1 will be the first row in this order that adheres to the selection condition c. Similarly, the correct x_2 corresponding to y_2 will be the last row in this order that adheres to the selection condition c. Thus, searching for x_1 and x_2 should begin either at the start, or the end of the sorted x attribute. In case the slope a is negative, this is reversed. The correct x_1 corresponding to y_1 will be found in the last row selected via the condition c and the correct x_2 corresponding to y_2 will be in the first row selected via the condition c. Since we do not know the actual sign of the slope, we propose to choose consecutive x_1 values in the outside-in order.

The above discussion is summarized in Algorithm 1. In line 1, the attributes x and y are extracted and sorted into arrays X and Y. We also assume that only unique values are retained. In line 2, the extreme values of the y attribute are retrieved as y_1 (the smallest) and y_2 (the biggest). In line 3, two pointers s and e are initialized. The pointer s points to the start of the X array, while the pointer e points to the end. The pointers are used to implement the outside-in order. Each iteration of the while loop (lines 4-19), corresponds to two x_1 values stored in X at pointers s and e. At the end of each iteration, the pointer s is incremented while the pointer e is decremented (line 18). The while loop ends when $s > e$. A for loop, defined in lines 5-17, is responsible for iterating over these two pointers. In line 6, the value x_1 is retrieved. Now, it must be paired with every other value in the X array. This is performed by the for loop in lines 7-16. To increase the chance of finding the correct x_2 value as soon as possible, we start from the end of the X array if the s pointer is used, or from the start of the array X if the e pointer is used. The value x_2 is retrieved in line 8. If $x_1 \neq x_2$, the slope a and the intercept b are computed (lines 10 and 11). Next, we verify if every value in Y can be computed based on some value in X.

The verification algorithm is described in Section 3.3. If the verification is positive, then the computed a and b values are returned as a result (line 13). If no result was found and the while loop ends, then there is no relationship between the tables S and T via attributes x and y. The slope a and intercept b are returned as null values (line 20).

3.3. Verification

The verification step determines whether every value in attribute y can be derived from values in attribute x using the current slope a and intercept b. In general, this is a problem of verifying whether one set is a subset of the other, complicated by floating point rounding errors.

Given that the arrays X and Y are sorted, three algorithmic approaches are feasible: (1) a sorted join-like approach ($O(|X| + |Y|)$), (2) iterative binary search ($O(|Y| \log |X|)$), and (3) hashing ($O(|X|)$

Algorithm 1 Full search for slope a and intercept b.

1: $X \leftarrow \tau_x(\Pi_x(S)), Y \leftarrow \tau_y(\Pi_y(T))$
2: $y_1 \leftarrow Y[0], y_2 \leftarrow Y[|Y|-1]$
3: $s \leftarrow 0, e \leftarrow |X|-1$
4: **while** $s \leq e$ **do**
5: **for** $i_1 \in \{s, e\}$ **do** ▷ **if** $s \neq e$
for $i_1 \in \{s\}$ **do** ▷ **if** $s = e$
6: $x_1 \leftarrow X[i_1]$
7: **for** $i_2 = |X|-1, |X|-2, \ldots, 0$ **do** ▷ **if** $i_1 = s$
for $i_2 = 0, 1, \ldots, |X|-1$ **do** ▷ **if** $i_1 = e$
8: $x_2 \leftarrow X[i_2]$
9: **if** $x_1 \neq x_2$ **then**
10: $a \leftarrow (y_1 - y_2)/(x_1 - x_2)$
11: $b \leftarrow y_1 - ax_1$
12: **if** $verify(X, Y, a, b, \epsilon)$ **then** ▷ Alg. 2
13: **return** a,b
14: **end if**
15: **end if**
16: **end for**
17: **end for**
18: $s \leftarrow s+1, e \leftarrow e-1$
19: **end while**
20: **return** $null, null$

initialization plus $O(|Y|)$ verification). Regarding the complexities, it is also worth noting that they represent a situation in which all values in the y attribute find a match in the x attribute. Since a single y value without a match is sufficient to abort the verification, in most cases the cost of this step will be much lower. We have experimentally evaluated the first two methods. Preliminary experiments indicated that the sorted-join like approach was significantly slower than the binary search alternative. Consequently, the experimental section reports results only for the binary search variant.

For clarity and reproducibility, we detail the binary search variant in Algorithm 2. The algorithm requires the sorted arrays X and Y (from Algorithm 1), the linear parameters a and b, and a tolerance threshold ϵ. The tolerance parameter is crucial for handling floating-point inaccuracies. Two values are considered equal if their absolute difference is less than or equal to ϵ. The algorithm starts with the initialization of the $start\_idx$ variable, which represents the lower bound of the search range within array X. The main loop (lines 3-20) iterates through Y. In each iteration, we calculate the theoretical value v (line 4) and determine if a corresponding value exists in X. Line 5, performs a binary search within the range $[start\_idx, |X|-1]$ to find the first insertion point for v that maintains the sorted order of X. Lines 7-12 and 13-18 check if the values in X at indices idx or $idx-1$ fall within $v \pm \epsilon$. If neither satisfies the condition, the parameters a and b fail to explain the dataset, and the function returns $false$ (line 19). Conversely, upon a successful match, $start\_idx$ is updated to the current position to narrow the search space for subsequent iterations (lines 9 and 15). If all corresponding x values are found, the function returns $true$ (line 21).

3.4. Reconstructing selection conditions

For the purpose of the discussion, let us briefly assume that the x attribute in relation S stores unique values. Let us also assume that the algorithm described in Section 3.2 has found the slope a and the intercept b.

In order to determine the selection condition c, we must first mark the selected rows. We need to compute an additional attribute in S such, that for each row it stores either 1 or 0, depending on

Algorithm 2 Verification of the slope a and the intercept b

```
procedure Verify(X, Y, a, b, ε)
    start_idx ← 0
    for y ∈ Y do:
        v ← (y − b)/a
        idx ← BinarySearch(v, X, [start_idx, |X| − 1])
        is_match ← false
        if idx < |X| then
            if |X[idx] − v| ≤ ε then
                start_idx ← idx
                is_match ← true
            end if
        end if
        if ¬is_match and idx > 0 then
            if |X[idx − 1] − v| ≤ ε then
                start_idx ← idx − 1
                is_match ← true
            end if
        end if
        if ¬is_match then return false
    end for
    return true
end procedure
```

whether the row was selected via the condition c or not. Let us denote this attribute as d. Since we assume that the x attribute stores unique values, the rows can be selected via a simple join with the condition $ax + b = y$. Thus, the relation with the attribute d can be computed as follows:

$$\Pi_{s_1,\ldots,s_n,1 \text{ as } d}(\sigma_{ax+b=y}(S \times T)) \cup \Pi_{s_1,\ldots,s_n,0 \text{ as } d}(S \setminus \Pi_{s_1,\ldots,s_n,x}(\sigma_{ax+b=y}(S \times T))) \quad (5)$$

Note that the attribute x is replaced by the new attribute d.

The subsequent step involves identifying the selection condition c. By framing this as a classification problem, where $s_1, \ldots, s_n$ are the features and d is the target label, we can employ a decision rule model. Because this model's representation closely resembles relational selection conditions, the disjunction of the conditions of the rules yielding a positive classification (label 1) defines the selection condition c.

There are, however, several key differences that may cause standard classification algorithms to fail to retrieve the correct selection condition. In standard methodology, a dataset is divided into at least training and testing datasets. In our case, a testing dataset is unnecessary, as we only want to determine how rows were selected when relation T was created. We are not creating a model for future prediction. Moreover, traditional classification models avoid overfitting by creating simple rules that are allowed to cover a small number of examples contradicting the rule's label. While we also seek simple and non-redundant rules, they must exactly describe the training dataset. The rules must be pure.

In our initial experiments, the best results were achieved using a sequential covering algorithm (e.g., [29]) featuring two additional rule-set post-processing phases: rule merging and redundant rule subsumption. We do not claim novelty for these specific components, as the methods employed are well-established in the literature: (1) rule aggregation is similar to the RISE algorithm [30], while (2) rule subsumption is very similar to the last step of LEM2 procedure [31]. Nevertheless, we provide a brief description of the rule induction algorithm here to ensure clarity and enable the reproducibility of our results.

The main loop of rule induction is presented in Algorithm 3. It starts with the initialization of two sets: $rules$, which is the set of generated rules, and $remaining$, which represents the set of examples not yet

covered by generated rules (lines 1,2). The algorithm generates all rules for a single label, and only after all examples with that label are covered, it continues on to the next label. This order is represented by the for loop (lines 3-9) iterating over labels and the while loop (lines 4-8) iterating until every example with the current label is covered. In the inner loop, the rules for the current label are generated via the $GenerateRule$ function (line 5, see Algorithm 4) and appended to the $rules$ set (line 6). Finally, the $remaining$ set is updated by removing all the examples covered by the newly created rule (line 7). This initial step generates a set of rules that needs to be optimized. This optimization is performed via the $AggregateRules$ (line 10, see Algorithm 5) and $PruneRules$ (line 11, see Algorithm 6) procedures.

Algorithm 3 Rule induction

```
1:  rules ← ∅
2:  remaining ← all examples
3:  for label ∈ [1, 0] do
4:      while there is example with label in remaining do
5:          r ← GenerateRule(remaining, label)                ▷ Alg. 4
6:          rules ← rules ∪ r
7:          Remove examples covered by r from remaining
8:      end while
9:  end for
10: AggregateRules(rules)                                     ▷ Alg. 5
11: PruneRules(rules)                                         ▷ Alg. 6
```

Let us now describe the $GenerateRule$ function presented in Algorithm 4. The function starts with initializing the new rule with an always-true condition (line 2). This rule's condition is next updated with additional inequalities (conjunctions) until it covers only examples with the current $label$ (while loop in lines 3-6). The new inequalities are created in line 4. For simplicity, we do not provide the pseudocode, as it would take a lot of space, however, we describe this process in the following. At this step, multiple cuts (inequalities) are generated and the one with the best score is chosen. The score is computed as purity (fraction of covered examples with the correct $label$) multiplied by a large weight (10^9 in our implementation) plus the number of the covered examples. This means that in case two of the cuts have the same purity, we prefer the cut covering more examples. The possible cuts are generated for every feature, for every inequality operand (either $\leq$ or $>$) and for a subset of distinct values of the current feature (see [32]).

The best cut algorithm returns the feature, operand, and threshold combination with the highest score (which are appended to the rule in line 5) or indicates that no such result can be generated. In the latter case, the rule creation is finalized and the resulting rule is returned (line 7).

Algorithm 4 Rule creation

```
1: function GenerateRule(remaining, label)
2:     r ← "if true then label"
3:     while r is not pure in remaining do
4:         atr, op, thr ← best cut; break if not found
5:         add condition (atr, op, thr) to r
6:     end while
7:     return r
8: end function
```

After rules covering every example are generated, they must be optimized. The first optimization step is rule aggregation. We greedily aggregate rules while they still remain pure. This optimization step is presented in Algorithm 5. The algorithm starts with the initialization of the boolean $stable$ variable which indicates whether the set of rules changed in the last iteration of the while loop (lines 3-15). This loop iterates until no change is introduced to the rule set. While in this loop, for each combination of

two rules r_1 and r_2 with the same label (lines 5-14), we compute their union (line 7) and verify whether the union remains pure (line 8). If it does, than the rules r_1 and r_2 are replaced by their union in the rule set (line 9). The rule union is performed by taking the minimum of the two lower bounds and the maximum of the two upper bounds for every feature. In case some of the bounds are not defined, they are treated as $-\infty$ or ∞.

Algorithm 5 Rule aggregation

```
1:  procedure AggregateRules(rules)
2:      stable ← false
3:      while ¬stable do
4:          stable ← true
5:          for r1, r2 ∈ rules do                          ▷ All combinations
6:              if label of r1 equals label of r2 then
7:                  r ← union of the rules r1 and r2
8:                  if rule r is pure then
9:                      rules ← rules \ {r1, r2} ∪ {r}
10:                     stable ← false
11:                     break
12:                 end if
13:             end if
14:         end for
15:     end while
16: end procedure
```

The final rule optimization step is the removal of redundant rules, presented in Algorithm 6. The algorithm starts with the initialization of the new rule set (line 2). Next, each rule, generated so far, is checked for redundancy (for loop in lines 3-10). This is done by finding the set of examples covered by the rule and the set of examples covered by all other rules with the same label (lines 4-6). If the set of examples covered by the rule is not a subset of the examples covered by other rules (line 7) then it is not redundant and is stored in the new, final, rule set (line 8). The set with non-redundant rules is returned in line 11.

Algorithm 6 Rule pruning

```
1:  procedure PruneRules(rules)
2:      result ← ∅
3:      for r ∈ rules do
4:          R ← rules ≠ r with label of r
5:          E_r ← examples covered by r
6:          E_R ← examples covered by R
7:          if E_r ⊈ E_r then
8:              result ← result ∪ {r}
9:          end if
10:     end for
11:     rules ← result
12: end procedure
```

We now return to the assumption regarding the uniqueness of values in attribute x of relation S. In general, this assumption does not hold. Consequently, multiple rows in S may contain values in x that correspond to a single value in attribute y. If only a subset of these rows was selected when the relation T was created, it may result in the inclusion of incorrect rows in the dataset used for rule generation. In order to avoid this problem, examples corresponding to non-unique x values should be pruned beforehand.

3.5. Limitations and future work

In this section, we discuss the limitations of the proposed solution.

The first limitation is related to the possible ambiguity when finding the linear parameters a and b. Consider a situation where relation S has 5 rows with x values in the set $\{-4, -1, 0, 1, 4\}$, while relation T has 3 rows with x values in the set $\{1, 2, 3\}$. In such a case, the following functions would pass verification:

- $y = x + 2$
- $y = -x + 2$
- $y = 0.25x + 2$
- $y = -0.25x + 2$

The problem is that there are several mappings of y values to x values such that the relative distances between consecutive x and consecutive y values are proportional. Algorithm 1 aborts the search after the first a, b pair passes verification successfully. However, it can be easily modified to continue computations until all possibilities are exhausted. In order to determine which function is the original one (in case many are found), expert analysis of the solution, as well as the reconstructed selection conditions, might be needed. In the future, we plan on devising automatic methods that solve this problem.

The second limitation stems from the problem already mentioned in Section 3.4. If values in attribute x are not unique, then there might be a problem with choosing exemplary rows for rule induction. We suggested to remove all rows corresponding to duplicate x values from the training dataset. However, if too many examples are removed, the reconstruction of selection conditions might become impossible. In the future, we plan to investigate methods for preserving some of the examples that would otherwise be removed.

The third limitation stems from the complexity of Algorithm 1, which requires $O(|S|^2|T|\log(|S|)$ in the worst case, i.e., in case where most verifications fail after checking a large portion of the Y array and there is no solution, so all possibilities need to be exhausted. However, in the experiments we noticed that verifications abort quickly, so the complexity is actually lower. Moreover, as mentioned in Section 3.2, it is suggested to first perform the Kolmogorov-Smirnov test to determine whether the solution can possibly exist before actually executing this algorithm. Finally, since many of the computations are independent, it is possible to parallelize Algorithm 1. This will be the subject of our work in the near future.

Unfortunately, even if a solution exists, numerous iterations of the outer loop in Algorithm 1 might be needed before it is found. To solve this problem, in our future work we plan on adapting the approach we presented in [7] to roughly estimate a and b. Based on these estimations, it is possible to pinpoint the corresponding indices i_1 and i_2 used in Algorithm 1, and start the search in their neighborhood. This will allow us to find a solution faster if it exists, with dismissible degradation of performance otherwise.

The final limitation of our approach stems from the assumed selection condition c. As mentioned in Section 3.2, condition c is represented in disjunctive normal form, where each clause is a conjunction of literals, and each literal is an inequality of the form $s_i\{\leq, >\}v$, where v belongs to the domain of s_i. However, it is easy to imagine conditions that involve multiple attributes in a comparison, e.g., $s_1 + s_2 < s_3$. Unfortunately, the adopted rule induction algorithm cannot discover such conditions. The final topic of our future work is therefore designing a new algorithm for selection condition reconstruction, capable of finding conditions that compare linear combinations of attribute values with an arbitrary threshold.

4. Experiments

To validate the proposed algorithms for slope and intercept estimation (Algorithm 1) and the reconstruction of selection conditions (Algorithm 3), we performed a series of experiments using synthetic data to simulate various transformation scenarios and logical conditions.

We begin with the slope and intercept estimation. As Algorithm 1 requires only attributes x and y, only these were generated. The data generation process begins by initializing a source set S with the given number of values drawn from a uniform distribution, which are then sorted in ascending order. A derived subset, T, of cardinality cT is then constructed through a sampling process that strictly enforces the inclusion of elements at both the lower bound index i_1 and the upper bound index i_2. The remaining $cT - 2$ values are drawn randomly without replacement from the index interval (i_1, i_2). Following the selection, a linear transformation defined by the slope and intercept is applied to the elements of T. Subsequently, both datasets are independently shuffled to eliminate the ordinal structure introduced during the initial phase.

As mentioned in Section 3.5, the complexity of Algorithm 1 with verification using Algorithm 2 is $O(|S|^2|T|\log(|S|))$. The $|S|^2$ term stems from the nested loops in Algorithm 1, while the $|T|\log(|S|)$ term represents the complexity of verification. Regarding the nested loops in Algorithm 1, the inner for loop in almost every iteration of the outer while loop is executed completely. Thus, its processing time depends primarily on $|S|$ and the verification cost. Total processing time depends on the number of iterations of the outer loop. In the worst case, this number is equal to $|S|$, which yields the aforementioned worst-case theoretical complexity. However, if a result is found earlier, the loop terminates. In fact, the number of iterations of the outer loop depends on the position in the sorted x attribute (array X) of the match for the first value in y attribute (y_1 variable in Algorithm 1). This position corresponds to the i_1 parameter mentioned in the previous paragraph.

In order to verify the above analysis, we performed two experiments. In the first experiment we tested performance of the inner loop. We varied $|S|$ in range $10000, 15000, \ldots, 50000$ and $|T|$ was set to 25%, 50%, and 75% of $|S|$. The measured average processing times of the inner loop are shown in Fig. 1. As can be observed, the processing time of the inner loop is linear w.r.t. the size of S, but the size of T does not influence it significantly. This is mainly due to the early exit mechanism during verification. In our experiments, the verification for incorrect a and b failed almost immediately (after first or second y value to verify). This is confirmed by the results in Fig. 2, where we varied $|T|$ for several distinct values of $|S|$ and i_1, observing that the total processing time (not just the average inner loop time) remained constant.

The second experiment tested how the position i_1 influences the total processing time for constant $|S|$. We performed the experiment for three distinct sizes of S equal to 10000, 30000, and 50000 respectively. Regardless of the size of S, $|T|$ was kept constant and equal to 1000. For each series, we varied i_1 in the range $[0; |S| - |T|]$ with a step size of 2500 (adjusting the last step to account for $|T|$). The results are presented in Fig. 3. Since i_1 essentially determines the number of iterations of the outer loop, and the inner loop processing time is constant for constant $|S|$, the dependency is essentially linear. However, one can easily see the pyramid-like shape of the curves in the figure. This shape can be easily explained with the outside-in traversal order of the outer loop. Given this order, extreme values of i_1 are processed first (the least number of iterations, the shortest processing times), while values near the middle are processed last (the highest number of iterations, the longest processing times).

Table 1
Example results of the reconstructing selection conditions algorithm

#	Selection condition	Discovered condition	Decision
1	$a < 25 \vee b > 58$	$a \leq 25.04 \vee b > 57.95$	✓
2	$a > 16 \wedge b > 70$	$a > 16 \wedge b > 70.01$	✓
3	$(a < 26 \wedge b > 80) \vee c > 50$	$(a \leq 26.03 \wedge b > 80) \vee c > 50.01$	✓
4	$(a < 26 \wedge b > 77) \vee (c > 50 \wedge d < 15)$	$(a \leq 25.95 \wedge b > 77.02) \vee (c > 49.96 \wedge d \leq 15)$	✓
5	$a < 26 \vee b > 77 \vee c > 50 \vee d < 15$	$a \leq 26 \vee b > 77.38 \vee c > 49.57 \vee d \leq 15$	✓
6	$(a < 26 \vee b > 77) \wedge (c > 50 \vee d < 15)$	$(a \leq 25.92 \wedge c > 49.99) \vee (b > 77.03 \wedge d \leq 15.04)$ $\vee (a \leq 26 \wedge d \leq 15.04) \vee (b > 77.03 \wedge c > 51.27)$	✓
7	$a + b > 70$	A set of 38 disjuncts with a total of 77 inequalities	✗
8	$a * b < 1000$	A set of 306 disjuncts, two inequalities each	✗

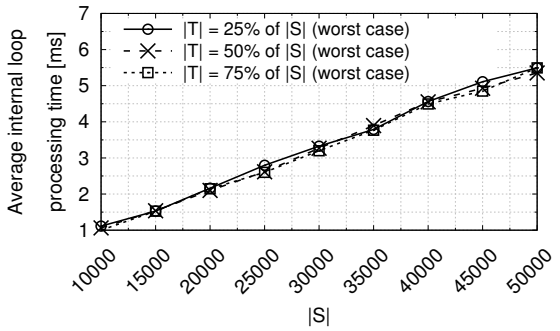

Figure 1: Average internal loop processing time in relation to the size of source relation $|S|$

We now discuss the evaluation of the selection condition reconstruction algorithm (Section 3.4). The data generation procedure was designed to simulate a real-world scenario where a target relation T is derived from a source relation S via the selection of tuples satisfying a specific condition, followed by the projection of a chosen attribute using a linear function. For each test, a source relation S consisting of $n = 10000$ tuples was generated. The number of attributes m in relation S varied from 3 to 5, depending on the complexity of the tested condition. The structure of relation S was defined as a set of attributes $\{s_1, s_2, \ldots, s_{m-1}, x\}$, where s_i act as conditional attributes, and x is the attribute subject to linear transformation. Attribute values were generated randomly according to a uniform distribution. For conditional attributes s_i, the value range was set to $[-100, 100)$, while for the attribute x, the range was $[-1, 1)$. The target relation T was created in two steps: (1) selection of tuples from relation S satisfying a defined condition, and (2) projection and transformation - for each selected tuple, the value of attribute x was transformed into value y using a linear function.

To validate the effectiveness of the discussed method, in Table 1 we provide example outcomes by contrasting the actual logical conditions with those recovered by the proposed approach. For clarity, attributes in the table are denoted by letters $a, b, c, \ldots$ rather than the s_i notation.

For selection conditions that adhere to the disjunctive normal form, the algorithm demonstrates very high effectiveness. The algorithm correctly identifies the logical structure of the query, including nested conjunctions ($\wedge$) and disjunctions ($\vee$). This is evident in row 6, where the complex logical structure was preserved in the discovered condition. Slight deviations in numerical values (e.g., 25 in the original vs. 25.04 in the discovered condition in row 1) result from the nature of the algorithm, which searches for "best cuts" between existing values in the dataset, as well as from floating-point precision issues. These differences are negligible and do not alter the semantics of the selection. Finally, the correctness of the recovered logical conditions serves as a validation for the first phase of our method, proving that the slope and intercept were successfully identified.

The last two cases (rows 7 and 8) illustrate the algorithm's behavior when encountering conditions that involve multiple attributes simultaneously. In these scenarios, the algorithm attempts to approximate the complex decision boundaries using a large number of disjuncts (and inequalities). While these selection conditions may be technically correct, they are rather useless for human interpretation or meaningful lineage tracking. However, it is crucial to emphasize, that these scenarios fall outside the scope of the problem definition established in Section 3.1. The initial assumptions explicitly constrained the selection conditions to disjunctive normal form. Therefore, rows 7 and 8 serve primarily to demonstrate the necessity of these assumptions rather than a failure of the algorithm within its intended domain.

5. Summary

In this paper, we present an algorithm capable of discovering relations that were created via selection and projection from source relations. The algorithm determines the linear function used to transform an attribute as well as reconstructs the selection condition used.

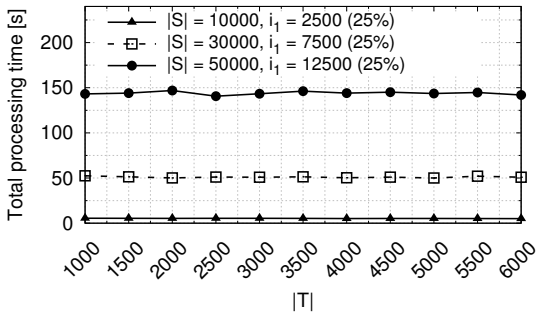

Figure 2: Total processing time as a function of the target relation size $|T|$

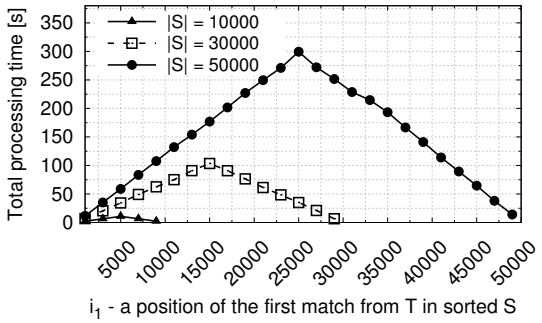

Figure 3: Impact of the first match position (i_1) in the sorted source S on the total processing time

Experimental results confirm the high accuracy of the reconstructed selection conditions (see Table 1), and consequently the precise recovery of the linear function's slope and intercept.

In future work, we plan to advance our research in two directions: (1) improving the performance of the linear function discovery algorithm, and (2) expanding the complexity of recoverable selection conditions. For details on future work, please refer to Section 3.5.

Declaration on Generative AI

The authors used Gemini 3 solely for grammar and spelling checks; promising content returned by the LLM was reviewed and rewritten as needed.

References

[1] T. Mucci, What is data provenance?, https://www.ibm.com/topics/data-lineage, n.d. IBM Think documentation; accessed Jan, 2026.

[2] D. P. Lanter, Design of a lineage-based meta-data base for gis, Cartography and Geographic Information Systems 18 (1991).

[3] What is data lineage?, https://www.ibm.com/topics/data-lineage, n.d. IBM documentation; accessed Jan, 2026.

[4] Y. Cui, J. Widom, Lineage tracing for general data warehouse transformations, VLDB Journal 12 (2003).

[5] Y. R. Wang, S. E. Madnick, A polygen model for heterogeneous database systems: The source tagging perspective, in: Int. Conf. on Very Large Data Bases (VLDB), 1990.

[6] M. Yamada, H. Kitagawa, T. Amagasa, A. Matono, Augmented lineage: traceability of data analysis including complex UDF processing, The VLDB Journal 32 (2023).

[7] W. Andrzejewski, P. Boiński, R. Wrembel, On fixing broken lineage, in: ProvenanceWeek @SIGMOD, 2025.
[8] P. Boiński, W. Andrzejewski, M. Grocholewski, T. Gruszczyński, R. Wrembel, Leveraging machine learning techniques for discovering broken lineage links between database objects, in: Information Systems Development (ISD), 2025. URL: https://doi.org/10.62036/ISD.2025.109.
[9] L. Chiticariu, W.-C. Tan, G. Vijayvargiya, DBNotes: a post-it system for relational databases based on provenance, in: Int. Conf. on Management of Data (SIGMOD), 2005.
[10] Y. Amsterdamer, D. Deutch, V. Tannen, Provenance for aggregate queries, in: ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS), 2011.
[11] D. Dosso, S. B. Davidson, G. Silvello, Data provenance for attributes: attribute lineage, in: USENIX Conf. on Theory and Practice of Provenance, TAPP, USENIX Association, 2020.
[12] J. N. Foster, T. J. Green, V. Tannen, Annotated XML: queries and provenance, in: ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems (PODS), 2008.
[13] T. J. Green, G. Karvounarakis, V. Tannen, Provenance semirings, in: ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), 2007.
[14] T. J. Green, V. Tannen, The semiring framework for database provenance, in: ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS), ACM, 2017.
[15] Y. Cui, J. Widom, J. L. Wiener, Tracing the lineage of view data in a warehousing environment, ACM Transactions on Database Systems 25 (2000).
[16] A. Kashliev, Storage and querying of large provenance graphs using NoSQL DSE, in: Int. Conf. on Big Data Security on Cloud, IEEE Int. Conf. on High Performance and Smart Computing, and IEEE Int. Conf. on Intelligent Data and Security (BigDataSecurity/HPSC/IDS), IEEE, 2020.
[17] R. Ikeda, J. Widom, Data lineage: A survey, Stanford University Publications (2009).
[18] P. Senellart, Provenance and Probabilities in Relational Databases, SIGMOD Record 46 (2018).
[19] M. Stamatogiannakis, P. Groth, H. Bos, Looking inside the black-box: Capturing data provenance using dynamic instrumentation, in: Int. Provenance and Annotation Workshop on Provenance and Annotation of Data and Processes, Springer-Verlag, 2014.
[20] J. a. F. Pimentel, J. Freire, L. Murta, V. Braganholo, A survey on collecting, managing, and analyzing provenance from scripts, ACM Computing Surveys 52 (2019).
[21] A. Chapman, P. Missier, G. Simonelli, R. Torlone, Capturing and querying fine-grained provenance of preprocessing pipelines in data science, VLDB Endowment 14 (2020).
[22] Y. Lou, M. Cafarella, Enabling useful provenance in scripting languages with a human-in-the-loop, in: Workshop on Human-In-the-Loop Data Analytics (HILDA) @SIGMOD, 2022.
[23] L. Moreau, J. Freire, J. Futrelle, R. E. McGrath, J. Myers, P. Paulson, The Open Provenance Model: An Overview, in: Provenance and Annotation of Data and Processes (IPAW), volume 5272 of *LNISA*, Springer, 2008.
[24] P. Missier, K. Belhajjame, J. Cheney, The W3C PROV family of specifications for modelling provenance metadata, in: Int. Conf. on Extending Database Technology (EDBT), 2013.
[25] PROV-Overview, https://www.w3.org/TR/prov-overview/, 2013. W3C Working Group documentation; accessed Jan, 2026.
[26] Q. T. Tran, C.-Y. Chan, S. Parthasarathy, Query reverse engineering, The VLDB Journal 23 (2014) 721–746. doi:10.1007/s00778-013-0349-3.
[27] C. Wang, A. Cheung, R. Bodik, Interactive query synthesis from input-output examples, in: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD '17, Association for Computing Machinery, New York, NY, USA, 2017, p. 1631–1634. doi:10.1145/3035918.3058738.
[28] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, V. Manquinho, SQUARES: a SQL synthesizer using query reverse engineering, Proc. VLDB Endow. 13 (2020) 2853–2856. doi:10.14778/3415478.3415492.
[29] P. Clark, T. Niblett, The cn2 induction algorithm, Machine Learning 3 (1989).
[30] P. Domingos, Rule induction and instance-based learning a unified approach, in: Int. Joint Conf. on Artificial Intelligence - Volume 2, Morgan Kaufmann, 1995.

[31] J. W. Grzymala-Busse, LERS-A System for Learning from Examples Based on Rough Sets, Springer, 1992.
[32] U. M. Fayyad, K. B. Irani, Multi-interval discretization of continuous-valued attributes for classification learning, in: Int. Joint Conf. on Artificial Intelligence, 1993.