## Summary of the Paper

### the concept

#### Inclusion Dependencies

Inclusion dependencies (INDs) are constraints that define relationships between attributes in a relational database. They specify that the values of one set of attributes must be included in the values of another set of attributes. In other words, if an inclusion dependency exists between two sets of attributes, the values of the first set must be a subset of the values of the second set.

For example, let's consider a database with two tables: "Customers" and "Orders". The "Customers" table has attributes like "CustomerID", "Name", and "Email", while the "Orders" table has attributes like "OrderID", "CustomerID", and "OrderDate". An inclusion dependency can be defined between the "CustomerID" attribute in the "Orders" table and the "CustomerID" attribute in the "Customers" table. This constraint ensures that the values of "CustomerID" in the "Orders" table must exist in the "CustomerID" attribute of the "Customers" table.

Inclusion dependencies are useful for maintaining data integrity and consistency in a database. They help ensure that the relationships between attributes are properly enforced and prevent inconsistencies or errors in the data. Database management systems often use inclusion dependencies to optimize query execution and enforce referential integrity.

In summary, inclusion dependencies define relationships between attributes in a database, specifying that the values of one set of attributes must be included in the values of another set. They are essential for maintaining data integrity and consistency in a relational database.

#### similarity Inclusion Dependencies

Similarity inclusion dependencies (sINDs) are a type of data dependency that express relationships involving the similarity between attribute values within or across relations. They can be seen as an extension of functional dependencies, which are used to describe relationships between attributes.

A similarity inclusion dependency is defined as follows:

Given two relations R(A) and S(B), where A and B are sets of attributes, a similarity inclusion dependency (sIND) is denoted as R[A] ⊑ S[B] and specifies that for every pair of tuples t1 ∈ R and t2 ∈ S, if the attribute values in t1 for the attributes in A are similar to the attribute values in t2 for the attributes in B, then t1 must also be in R.

In simpler terms, an sIND states that if two tuples have similar values for certain attributes, they must belong to the same relation. The notion of similarity varies depending on the context and can be defined using various similarity measures such as edit distance, cosine similarity, etc.

Here are a few examples of similarity inclusion dependencies:

Let's assume we have two relations, Customers(CustomerID, Name) and Orders(OrderID, CustomerName). We can define an sIND as Customers[Name] ⊑ Orders[CustomerName], which indicates that if two customer names are similar, they should correspond to the same customer ID.

Consider two relations, Products(ProductID, Description) and Recommendations(RecID, ProductDescription). An sIND can be defined as Products[Description] ⊑ Recommendations[ProductDescription], implying that if two descriptions of products are similar, they should represent the same recommended product.

Suppose we have two relations, Students(StudentID, Name) and Grades(GradeID, StudentFullName). We can specify an sIND as Students[Name] ⊑ Grades[StudentFullName], which states that if two student names are similar, they should refer to the same student ID.

These examples illustrate how similarity inclusion dependencies can be used to express relationships between attribute values based on their similarity.

### sINDs discovered by The Sawfish Algorithm

The Sawfish algorithm, which stands for Similarity AWare Finder of Inclusion dependencies via a Segmented Hash-index, is a data mining algorithm used to discover these sINDs in relational databases. It is designed to efficiently identify relationships between columns in tables based on their similarities.

#### A brief summary of the Sawfish algorithm

Preprocessing: In this step, Sawfish preprocesses the dataset by generating a set of metadata and creating a segmented hash index. The metadata includes information about the attributes and their corresponding domains.

Basic Discovery Approach: Sawfish uses a basic discovery approach to identify potential similarity inclusion dependencies (sINDs) in the dataset. It compares the values of different attributes and calculates their similarity scores.

Inverted Index: Sawfish constructs an inverted index based on the similarity scores. This index helps identify the possibly similar strings by storing pointers to the tuples that have similar attribute values.

Dependent Value Validation: To validate the sIND candidates, Sawfish uses two modes - ED mode (Edit Distance mode) and JAC mode (Jaccard similarity mode). In ED mode, it calculates the edit distance between attribute values to determine similarity, while in JAC mode, it calculates the Jaccard similarity coefficient.

sIND Discovery: Using the inverted index and similarity validation modes, Sawfish performs the actual sIND discovery. It evaluates the sIND candidates and verifies if they satisfy the similarity inclusion dependencies. If a candidate satisfies the conditions, it is considered a discovered sIND.

That’s the general process of similarity inclusion dependencies (sINDs) discovery based on the Sawfish algorithm. Please note that the details of the algorithm and its implementation may vary based on the specific Sawfish implementation and the dataset used.

#### Examples of the Sawfish algorithm in action

##### a sample dataset

Dataset:

EmployeeID Name Department Salary Age

1 John Smith HR 5000 30

2 Jane Doe IT 6000 35

3 Mike Johnson IT 5500 40

4 Amy White HR 4500 28

5 Jack Black HR 5500 32

6 Sarah Adams IT 5200 31

| **EmployeeID** | **Name** | **Department** | **Salary** | **Age** |
| --- | --- | --- | --- | --- |
| 1 | John Smith | HR | 5000 | 30 |
| 2 | Jane Doe | IT | 6000 | 35 |
| 3 | Mike Johnson | IT | 5500 | 40 |
| 4 | Amy White | HR | 4500 | 28 |
| 5 | Jack Black | HR | 5500 | 32 |
| 6 | Sarah Adams | IT | 5200 | 31 |

Algorithm Execution Process:

Preprocessing: The metadata includes information about the attributes (EmployeeID, Name, Department, Salary, and Age) and their corresponding domains.

Basic Discovery Approach: In this case, it will compare the values of attributes like Name, Department, Salary, and Age to determine if there are any potential similarities.

Inverted Index: In this case, for example, it will create an inverted index linking similar names or similar department values across different tuples. inverted indexes for the attribute pairs (EmployeeID, Name), (EmployeeID, Department), (EmployeeID, Salary), (EmployeeID, Age), (Name, Department), (Name, Salary), (Name, Age), (Department, Salary), (Department, Age), and (Salary, Age).

Dependent Value Validation: In this case, for example, it will calculate the edit distance between different Names or the Jaccard similarity between different Departments. Let’s assume we want to find the top 2 most similar values for each attribute pair.Compute the similarity between values of each attribute pair and store the top 2 similar values in the corresponding inverted index.Consider the pair of attributes (EmployeeID, Name) and their top 2 similar values:1, John Smith) and (2, Jane Doe).Check if the set of EmployeeIDs corresponding to John Smith is a subset of the set of EmployeeIDs corresponding to Jane Doe based on the similarity measure.If the condition holds true, output the sIND: EmployeeID ⊆ Name.Repeat the same process for all other attribute pairs.

sIND Discovery: In this case, for example, it will check if the tuples with similar Names or Department values have similar Salary or Age values. If a candidate satisfies the conditions, it is considered a discovered sIND.Based on the provided dataset, the Sawfish algorithm may discover potential sINDs such as:

Employees with similar names may have similar salaries or ages.

Employees in the same department may have similar salaries or ages.

Please note that the actual discoveries depend on the similarity thresholds set and the specific implementation of the Sawfish algorithm. The above examples are just illustrative and may not reflect the real sINDs that would be discovered in practice.

Here are a few examples to explain the Validation and Discovery process in detail.

###### Example1:

Algorithm 1 (ED mode) Example:

Suppose we want to check for similar strings for the value “Jon Simth” (which is a similar string to “John Smith”).

Algorithm Execution Process:

Step 1 - ProbeIndex Function:

Set the input value to “Jon Simth”. Initialize the ProbeIndex function with the given parameters (indices and the target value, “Jen Smith”).

Step 2 - Loop over ld:

Determine the LD (Levenshtein Distance) range based on the input value and the parameter τ (used in the algorithm).

If τ = 1, the LD range will be [-1, 1]. (which is a predefined threshold).

Step 3 - Generate Substrings:

Generate all possible substrings within the LD range.

In this case, the generated substrings for ““Jon Simth” are:

Substrings[-1]: [“Jon Simt”, “Jon Sim”, “Jon Si”, “Jon S”, “Jon”]

Substrings[0]: [“Jon Simth”, “Jon Simt”, “Jon Sim”, “Jon Si”, “Jon S”, “Jon”]

Substrings[1]: [“Jon Simth”, “Jon Simt”, “Jon Sim”, “Jon Si”, “Jon S”]

Step 4 - Get Index for the Current ld Value:

Retrieve the index for the given ld value from the indices.

For example, if indices[-1] refers to index 1, then we retrieve index 1 for ld = -1.

Get the index corresponding to the input value length + LD (Levenshtein Distance).

Step 5 - Initialize alreadyValidated Set:

Create an empty set called alreadyValidated, which will store and keep track of the already validated matches.

Step 6 - Iterate over Substrings and Index Segments:

Iterate through each index (segment) within the LD range.

Step 7 - Get SegmentMap:

Retrieve the segmentMap for the current index segment at position i.

Step 8 - Matches Set:

Create an empty set called matches to store the matches found during the iteration.

Step 9 - Iterate over Substrings for Each SegmentMap:

Iterate over each substring for the current index segment.

Step 10 - Iterate over Matches for Each Substring:

Iterate over each match found in the segmentMap for the current substring.

Step6~10 Execution:

For i = -1:

segmentMap = index[-1]

No matches in this segment.

For i = 0:

segmentMap = index[0]

No matches in this segment.

For i = 1:

segmentMap = index[1]

Matches found: “John Smith” (EmployeeID: 1)

Step 11 - Check if Match is Already Validated:

Check if the current match is not already present in the alreadyValidated set.

Step 12 - Add to Matches and alreadyValidated:

If the match is not already validated, add it to the matches set and the alreadyValidated set.

Step 13 - Update alreadyValidated:

Update the alreadyValidated set with the newly added match.

Step 14 - Check if Matches is not Empty:

If the matches set is not empty, proceed to the next step. Otherwise, continue to the next iteration.

Step 11~13 Execution:

Check if the matches from each segment are not already validated.

Matches from segment 1: “John Smith” (EmployeeID: 1)

Store the valid matches.

Matches: “John Smith” (EmployeeID: 1)

Add the validated matches to the alreadyValidated set.

alreadyValidated: {“John Smith” (EmployeeID: 1)}

Step 15 - Validate Matches:

Since valid matches are found, proceed to validate the matches. Validate the matches using a validation function (not specified in the given algorithm).

Pass the value “Jen Smith” and the matches set to the ValidateMatches function to validate the matches.

Step 16 - Check if existsSimilarString is True:

If the result of the ValidateMatches function is True(the matches are validated as similar), set the existsSimilarString flag as True.

Step 17 - Return existsSimilarString:

If existsSimilarString is True, return it as the result.

If existsSimilarString is True, return existsSimilarString.

Step 18 - Return False:

If no similar string is found or validated after iterating through all segments, return False as the result.

In this sample execution process, Algorithm 1 Index Probing in ED mode with the given dataset does not find any similar string for the input value “Jon Simth”.

Algorithm 2 (JAC mode) Example:

Let’s assume we want to search for a similar string “John Doe” in the dataset.

Algorithm Execution Process:

Set the input value to “John Doe”.

Determine the index length range based on the value length and δ (a parameter used in the algorithm).

If δ = 0.5 and |value| (length of the value) = 8, the index length range will be [4, 6] (ceil(0.5 \* 8) = 4, floor(8 \* 0.5) = 4).

Get the index corresponding to the selected index length.

Calculate the threshold T using the formula T = δ / (1 + δ) \* (|value| + indexLength).

If δ = 0.5, |value| = 8, and indexLength = 4, T = 0.5 / (1 + 0.5) (8 + 4) = 2.

a count dictionary to store the frequency counts of matching tokens.

Tokenize the input value into individual tokens.

Tokenized tokens: [“John”, “Doe”]

Iterate over each token in the tokenized tokens.

For each token, for matches in the index.

“John” matches with “John Smith” (EmployeeID: 1) and “Johnson” (EmployeeID: 3).

“Doe” doesn’t match with any entry in the dataset.

Increase the count for each match found.

count[1] = 1 (match for “John Smith”)

count[3] = 1 (match for “Johnson”)

Check if the count for any match greater than or equal to the threshold T (2).

count[1] = 1 < 2

count[3] = 1 < 2

No match satisfies the condition count[match] ≥ T, so return False.

The algorithm finishes execution, and the result is False.

In this sample execution process, the algorithm does not find any similar string in the dataset based on the given input value “John Doe”.

###### Example2:

with Algorithm 1 Index Probing in ED mode and Algorithm 2 Index Probing in JAC mode to find similar strings and with Algorithm 3 Candidate Validation to find some valid unary sINDs:

Algorithm 1 (ED mode) Example:

Input value: “John Smit”

Substrings[0]: [“John Smith”, “John Smit”]

Substrings[1]: []

Substrings[2]: []

Matches found:

In substrings[0]:

“John Smith” (EmployeeID: 1)

The algorithm finds a similar string “John Smith” based on the given input value “John Smit”.

Algorithm 2 (JAC mode) Example:

Input value: “Jon Smth”

Index length range: [4, 6]

Index for length 4:

{“Jon”: [1]}

{“Smit”: [1]}

Matches found:

“John Smith” (EmployeeID: 1)

The algorithm finds a similar string “John Smith” based on the given input value “Jon Smth”.

In both cases, the algorithms are able to find similar strings in the dataset based on the given input values.

Algorithm 3 Candidate Validation

Assumption:

The columnBucketMap and τ (threshold) values are initialized during the preprocessing step.

Execution Process:

1、Initialize an empty set called ‘processed’ to keep track of processed columns.

2、Iterate while the size of ‘processed’ is not equal to the number of columns in the dataset.

3、Get the referenced columns for all columns in ‘processed’ using the columnBucketMap.

4、Start the loop for ‘l’ (column length) from 50 to 1.

5、Check if for all referenced columns, candidates[ref] is empty. If so, break the loop.

6、For each referenced column ‘ref’:

If candidates[ref] is not empty, proceed.

Initialize an empty set columnIndices[ref][l + τ + 1].

Build an index for columnref’ with length ‘l - τ’ using the columnBucketMap.

7、For each referenced column ‘ref’:

Get the indices from columnIndices[ref].

8、For each referenced column ‘ref’:

For each dependent column ‘dep’ in candidates[ref]:

For each value in columnBucketMap[dep]:

Probe the index using ProbeIndex(indices, value).

If a similar string does not exist, remove the dependent column ‘dep’ from candidates[ref] and break the loop.

If a similar string exists, continue.

9、For each referenced column ‘ref’:

For each dependent column ‘dep’ in candidates[ref]:

Output dep ⊂∼ ref (dependent column is not a subset of referenced column).

Using Algorithm 3 Candidate Validation, we can find some valid unary sINDs (keys and functional dependencies) based on this dataset. Here are a few examples:

EmployeeID -> Name (The employee ID uniquely determines the name of an employee.)

EmployeeID -> Department (The employee ID uniquely determines the department of an employee.)

EmployeeID -> Salary (The employee ID uniquely determines the salary of an employee.)

EmployeeID -> Age (The employee ID uniquely determines the age of an employee.)

These are just a few examples, and the actual valid unary sINDs (keys and functional dependencies) that could be discovered using Algorithm 3 Candidate Validation may vary depending on the specific dataset(values in the columnBucketMap) and the threshold (τ) used during the preprocessing step.

##### a sample dataset with two relations

Here’s a sample dataset with two relations, “Products” and “Recommendations”, that can be used with Algorithm 1 Index Probing in ED mode and Algorithm 2 Index Probing in JAC mode to find similar strings and with Algorithm 3 Candidate Validation to find valid unary sINDs:

Products:

ProductID Description

1 T-shirt

2 Jeans

3 Sneakers

4 Watch

Recommendations:

RecID ProductDescription

1 Upgrade your wardrobe with some new T-shirts.

2 Try out our latest collection of jeans.

3 Complete your look with stylish sneakers.

4 Get a fashionable watch to enhance your style.

Using Algorithm 1 Index Probing in ED mode and Algorithm 2 Index Probing in JAC mode, we can find similar strings between the “Description” attribute in the “Products” relation, and the “ProductDescription” attribute in the “Recommendations” relation.

Here are some examples of similar strings that can be found:

“T-shirt” and “Upgrade your wardrobe with some new T-shirt.”

“Jeans” and “Try out our latest collection of jeans.”

“Sneakers” and “Complete your look with stylish sneakers.”

Using Algorithm 3 Candidate Validation, we can find some valid unary sINDs (keys and functional dependencies) based on these two relations. Here are a few examples:

ProductID -> Description (The product ID uniquely determines the description of a product.)

RecID -> ProductDescription (The recommendation ID uniquely determines the product description in the recommendations.)

These are just a few examples, the actual similar strings and the actual valid unary sINDs that could be using algorithms may vary depending on the specific dataset and the threshold set during the preprocessing step.

Overall, the Sawfish algorithm provides an efficient and scalable approach to discovering sINDs in relational databases, enabling better understanding and analysis of the underlying data relationships.