An Acceleration of Inclusion Dependency Discovery and its Evaluation

Hiroaki UNO† Kazuhiro SAITO† ‡ and Hideyuki KAWASHIMA†

† Keio University 5322 Endo, Fujisawa-shi, Kanagawa, 252-0882 Japan

‡ KDDI Research, Inc. Garden Air Tower, 3-10-10, Iidabashi, Chiyoda-ku, Tokyo, 102-8460 Japan

E-mail: † hiroaki.uno@keio.jp, river@sfc.keio.ac.jp † ‡ ku-saitou@kddi-research.jp

Abstract In recent years, the variety of data is increasing, making data management tasks more and more complex. Data profiling is a process of determining metadata by extracting definitions of data from data itself. Inclusion dependency detection is one of the data profiling tasks which detects inclusion of data between columns, and is applied for discovery of relationship between data, and data integration. In this research, we propose an accelerated algorithm of accurate IND discovery, focusing on the two major components: candidate generation and IND discovery. The result showed that our proposals are effective when processing relations above a certain scale.

Keyword Data Profiling, Metadata, Inclusion Dependency, Data Structure

1. Introduction

In recent years, the variety of data is dramatically increasing. IoT devices and digital transformation have made it possible to generate various data by translating real world information into the data. In addition, open data and data trading are enabling us to utilize various external data that people except data owner could not touch before. We can now have an opportunity to utilize data for various purposes, such as high-quality data analysis and machine learning, by increasing a variety of such data.

However, more variation of data leads to more complex data management. One of the data management tasks in the environment where a wide variety of data is stored is data exploration that a user looks for data to utilize. Another example of the tasks is to find relationship between tables which can be joined and utilized together. More variety of data there are in the environment, more time is wasted for the tasks related to data management. In this situation, it is very important to create and collect information about data itself, called metadata, so that data can be managed more easily.

Data profiling[1] is a process of determining metadata by extracting data definitions from data itself, and it simplifies data management tasks. Data profiling can be classified into three tasks: Profiling for a “single column”, “multiple columns”, and “dependencies”. Inclusion dependency (IND) discovery is one of the important profiling tasks which find dependencies between columns including dependencies across relations. An IND states that all tuples of some columns in a relation are included in some other columns in the same or a different relation[2]. The IND discovery is applied for foreign key discovery. Furthermore, it can be ranked as a prerequisite for data integration.

The objective of our research is to accelerate discovery of inclusion dependencies. In this research, we focus on exact discovery algorithms. Exact algorithms aim to guarantee the completeness of the result; in IND discovery context, the exact algorithms try to find every INDs and only INDs. In contrast, approximate algorithms pursue time-efficiency while the completeness and accuracy of the result are not always guaranteed; approximate discovery algorithms can return results with not-INDs (false positives) or results missing some INDs (false negatives). Without any acceleration method, the complexity of exact discovery of IND grows hyper exponentially with arity. We propose a time-efficient algorithm, both referring to existing IND discovery algorithms and introducing other improvements.

The remainder of this paper is organized as follows: In chapter 2, the general flow of the multiple column IND discovery is explained. Chapter 3 explains our proposals to two major components of IND discovery algorithm. In chapter 4 we explain the method of evaluation and show results. In chapter 5 we discuss the results and further improvements. After mentioning the existing algorithm in chapter 6, we conclude in chapter 7.

2. Overview of IND discovery

We explain the overview of IND discovery in this chapter. In a relation R, let $|R|$ the number of columns, and $|r|$ the number of tuples. Here, a column combination is defined as one or more columns selected from all columns $r_1, r_2, \ldots r_{|R|}$ in R. A column combination a in R is expressed

$R[a]$. Given two relations R and S, if values in any tuple of column combination $R[a]$ are a subset of values of column combination $S[b]$, there exists an inclusion dependency of $R[a]$ on $S[b]$. The dependency is written as $R[a] \subseteq S[b]$. Furthermore, the left-hand side $R[a]$ is called the dependent column combination and right-hand side $S[b]$ the referenced column combination. We say that both sides together comprise "a pair of column combination".

The common flow of multiple column IND discovery can be described as the following. (Also shown in エラー! 参照元が見つかりません。.) We can see that in each arity, the algorithm is comprised of two kinds of process, candidate generation and IND checking.

Algorithm 1: Multiple column IND discovery

```
Input: $R$, $S$: relations
Output: $INDlist$: List of INDs from every arity
$INDlist$ = null;
$arityUpperlimit$ = $min\{|R|, |S|\}$;
for $arity$ = 1 to $arityUpperlimit$
    $candidateList$ = $gen$Candidates($arity$, $R$, $S$);
    for each $candidate$ from $candidateList$
        $isIND$ = checkIND($arity$, $candidate$, $R$, $S$);
        if $isIND$ is TRUE
            add $candidate$ to $INDlist$
return $INDlist$
```

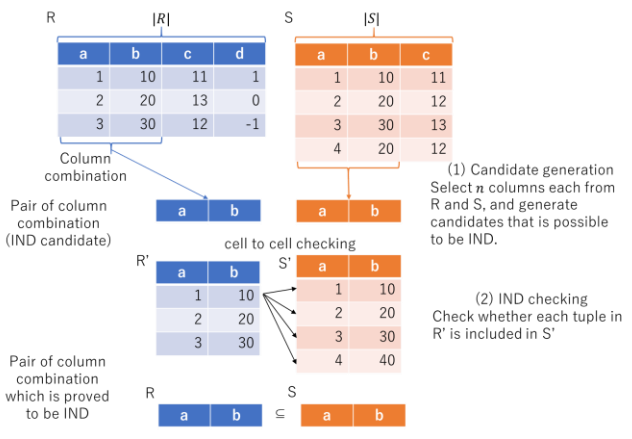
R
|R|
S
|S|
Column combination
Pair of column combination (IND candidate)
(1) Candidate generation
Select n columns each from R and S, and generate candidates that is possible to be IND.
cell to cell checking
R'
S'
(2) IND checking
Check whether each tuple in R' is included in S'
Pair of column combination which is proved to be IND
⊆

Figure 1
Flow of multiple column IND discovery

3. Proposal

In this chapter, we state possible acceleration methods to two major components of IND discovery: candidate generation and IND checking.

3.1. Candidate generation algorithm

Candidate generation in n-ary IND discovery includes selecting n columns each from relation R and S, generating pairs of column combination which has a possibility to be dependent and referenced ones in inclusion dependency.

3.1.1. Naïve method

The naïve method generates IND candidates independently for each arity. The procedure in n-ary is as follows.

(1) Generate n-ary combinations without replacement from columns in R, making dependent sides of IND.
(2) Generate n-ary combinations without replacement from columns in S, making referenced sides of IND.
(3) Get pairs of column combinations by generating the Cartesian product of dependent and referenced sides.

The reason for applying different ways of generating column combination (combination for R, and permutation for S) is to generate IND candidates in exact quantity.

If permutation is applied both for R and S, for example, duplicate candidates will be generated with substantially identical meaning. For example, $R[A, B] \subseteq S[A, B]$ and $R[B, A] \subseteq S[B, A]$ have different expressions but are identical as IND candidates.

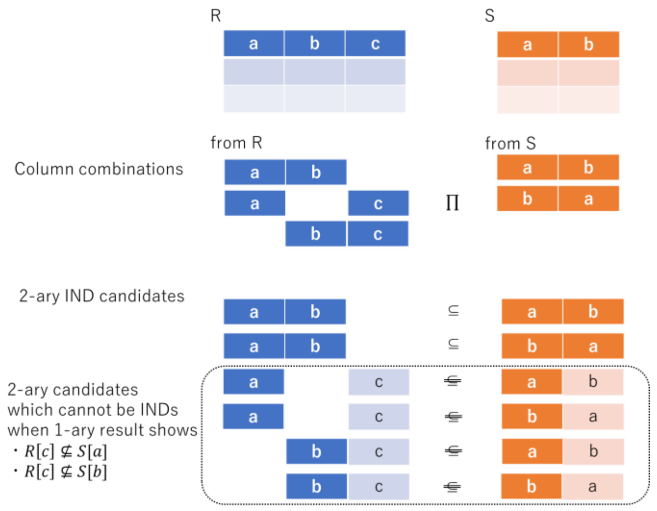

Figure 2 n-ary naïve candidate generation
Example shown in 2-ary. The dotted rectangle shows 2-ary candidates which cannot be INDs.

The complexity of naïve candidate generation can be estimated as follows.

$\left({}_{|R|}C_n \times n!\right) \times \left({}_{|S|}P_n \times n!\right)$ **when** $n \leq min\{|R|, |S|\}$

This naïve method does not require incremental search from unary. It means that this method does not reflect the result of actual INDs of lower arities. This leads to a possibility of generating false positive candidates which are not actually INDs. For example, even when $R[c] \subseteq S[b]$ is not true in unary, the method generates candidates such as $R[a,c] \subseteq S[a,b]$ in 2-ary.

3.1.2. Fast Incremental method

We propose an incremental version of candidate generation, the Fast Incremental method. Its procedure is as follows.

In case of 1-ary, the algorithm utilizes the naïve method. Note that in unary generating permutations and combination is equal

(1) Generate dependent sides by selecting one column each from R.
(2) Generate referenced sides by selecting one column each from S.
(3) Get IND candidates by generating Cartesian product of both sides.

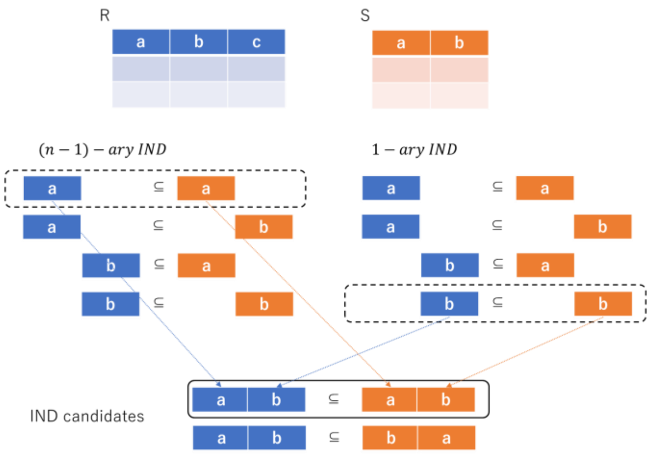

Figure 3 n-ary candidate generation based on (n-1)-ary and 1-ary IND
Example shown in 2-ary.

In case of 2-ary or more, the algorithm employs the result of IND checking of lower arities.

(1) Generate n-ary candidates by combining $(n-1)$-ary and unary INDs. Combine dependent side (left-hand side, R) of unary to that of $(n-1)$-ary, and referenced side (right-hand side, S) respectively. (Shown in エラー! 参照元が見つかりません。.)
(2) The combined result is only employed only if the following two conditions are fulfilled:
   i. For dependent side, the column from unary IND is rightmost in original R compared to columns from $(n-1)$-ary IND.
   ii. For referenced side, each column should be unique.

Through this algorithm, candidates which cannot be IND are pruned and this contributes to reduction of the calculation cost in IND checking phase.

3.2. IND checking algorithm

IND checking is a process of determining whether each generated candidate is actually IND (whether the dependent column combination is included in referenced one) by scanning tuples.

In IND checking, we define complexity as the frequency of checking cell to cell in relation.

3.2.1. Naïve method

The naïve IND checking algorithm employs the method similar to nested loop join. The algorithm checks all the tuples in referenced side for each tuple in dependent side.

The complexity, the frequency of checking cell to cell, is $|r| \times |s| \times n$.

3.2.2. Hashing with PLI

We incorporate a technique of acceleration by hashing the referenced side relation. This corresponds to the one-way hash join.

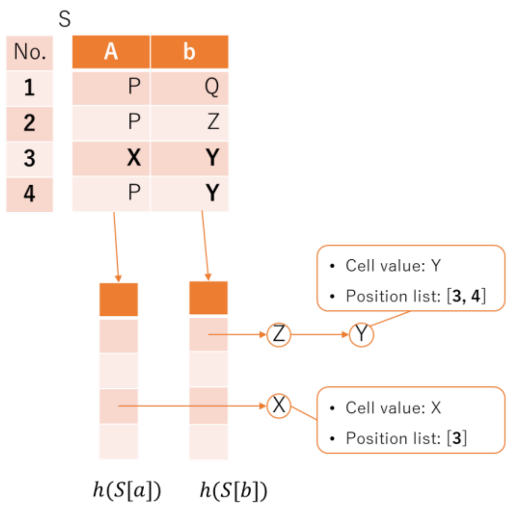

Figure 4 Hash table incorporating PLI
The values X and Y are stored in different hash tables. Both X in h(S[a]) and Y in h(S[b]) has same 3 in their position

lists, thus IND checking program can judge that X and Y are in the same tuple in original relation S.

In our method, hash tables are generated independently from each column just after loading the relation so that the algorithm does not have to regenerate hash tables for each arity. At this moment, $|S|$ distinct hash tables are generated. Each node in a hash table has a Position List Index (PLI)[3] data structure, and can contain not only the value of the column, but also the tuple number(s) where the value is from. (Shown in エラー! 参照元が見つかりません。.)

Storing tuple numbers is to check whether values hashed in different hash tables are contained in the same tuple or not in the original referenced relation, when checking INDs of 2-ary or more.

When checking n-ary INDs, the procedure retrieves the cell value of R from hash table of corresponding column of S. Since column combination has n columns, the procedure is repeated n times for each tuple in R. Thus, the complexity is reduced to $|r| \times n$.

4. Evaluation

In this chapter, we compare and evaluate the performances of IND discovery algorithms.

4.1. Environment

We explain briefly about implementation, followed by the methods and conditions of evaluation.

4.1.1. Implementation

We implemented the IND discovery algorithms in C language. The program accepts two CSV files. Each CSV file represents a single relation. The program checks if there are any INDs from the first relation to second relation. The first CSV file is a potentially dependent relation and second a potentially referenced relation.

4.1.2. Methods of evaluation

We conduct two types of experiments; we compare and evaluate the IND program from two different ways.

In the first experiment, we check how the execution time grows as the arity grows. This is to see the effect of different candidate generation methods.

Another experiment checks how the execution time grows as records in a relation increase in number. This is to see the difference between IND checking algorithms.

4.1.3. Details of the experiments

We prepared CSV files which contain pseudo-random integer values for the experiments.

In the test procedure, we designate the same CSV file for dependent and referenced relation as input for the program. This is because our test aims to measure the performance of IND programs. Since all the methods we introduced in this paper are accurate algorithms, we do not evaluate the accuracies of each method.

We execute the same-setting trial for five times, and measure execution time each, and calculate the average of execution time.

4.1.4. Conditions of the experiments

We conducted experiments under the following conditions.

Hardware

- CPU: Intel Core i5-7200U 2.50GHz 2.71GHz
- RAM: 8.00GB
- ROM: 256GB

Software

- OS: Windows 10 64bit
- C Compiler: realgcc.exe (Rev1, Built by MSYS2 project) 7.2.0
- Shell: GNU bash, version 4.4.19(2)-release (x86_64-pc-msys)

4.2. Result

The results of the experiments were as follows.

4.2.1. Comparison of candidate generation methods

We first show comparison of different candidate generation methods: the naïve method, and Fast Incremental method which we propose.

In this experiment conditions related to IND checking were set fixed as follows: The number of records in each relation was fixed constant to 100 records. Furthermore, naïve IND checking method was employed throughout this experiment.

Limit of arity	Naïve (average)	Proposal (average)
1	31.6	34.4
2	41	36.2

3	27	37.4
4	33.2	39.8
5	33.6	40.6
6	48.8	46.6
7	128.8	36.2
8	-	39
9	-	47.8
10	-	76.8

Table 1 Execution time (ms) of programs with different candidate generation methods

The blank (-) shows the result is unavailable.

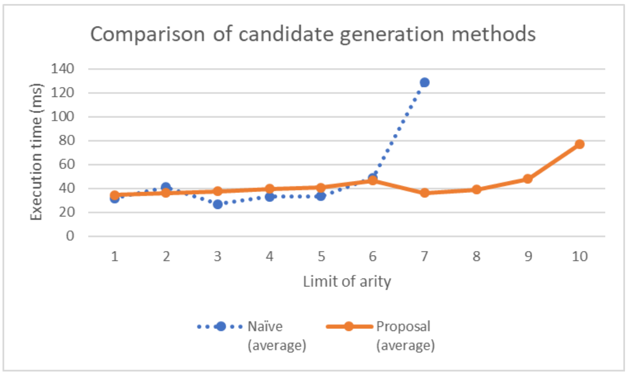

Figure 5
Comparison of averages of execution time with different candidate generation methods

4.2.2. Comparison of IND checking methods

We also compared different IND checking methods: the naïve method, and hashing with PLI.

While the number of records in the input files changes in this experiment, the conditions related to candidate generation were set fixed as follows: We designated relations with 6 columns as input, thus the limit of arity in the experiment is constantly 6-ary. Furthermore, the Fast Incremental method was employed throughout this experiment.

Number of records	Naïve (average)	Hashing with PLI (average)
1	59.2	118.8
5	61.6	63
10	48.2	60.2
50	72.4	56
100	71.2	69.8
500	105	88.8
1,000	208.2	108
5,000	3,155.8	1,252.8
10,000	16,299.4	7,564

Table 2 Execution time (ms) of programs with different IND checking methods

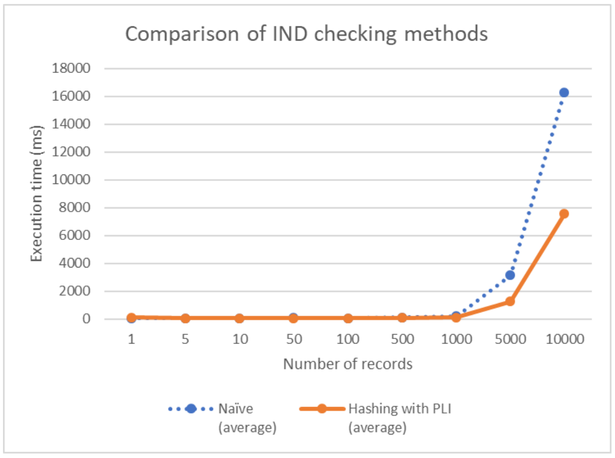

Figure 6
Comparison of averages of execution time with different IND checking methods

5. Discussion

In this chapter we discuss the result of the experiments and possible further improvements.

5.1. Interpretation of the evaluation

The results show that our proposals effectively reduce execution time in case arity or record size is above the certain level.

In the phase of candidate generation, our Fast Incremental method effectively reduce the execution time in 7-ary or more.

The result also shows the stability of our method. In 8-ary or more, naïve method could not complete the process. On the other hand, our method works properly at least in 10-ary or below.

It suggests that the Fast Incremental method effectively prunes the candidates, mitigating the exponential growth of candidates, leading to the fast and stable execution.

In the phase of IND checking, hashing with PLI effectively reduce the execution time when the number of records in a relation is more than 1,000.

Although the proposal method needs to prepare hash tables as an initial investment, the effect of hashing is expected to expand in actual environment as the number of records grow.

5.2. Future work

We raise following topics for further improvements in IND discovery.

First, incorporating other data profiling techniques will lead to more efficient candidate generation. Metadata of each column such as domain or column name is helpful in deciding whether column combinations from R and S should be paired. Applying these data profiling techniques before candidate generation is an option to be considered.

Second, we expect that parallel execution in the phase of IND checking would be effective. Whether a certain tuple in R matches a tuple in S does not affect the result of another tuple in R. This means that each tuple in R can be independently checked and parallel execution can be applied.

6. Related Work

Finally, we mention existing research which addresses IND discovery algorithms.

FAIDA[4] is an IND discovery algorithm published in 2017. It employs an overall framework in which IND candidates are generated based on actual INDs of lower arities.

FAIDA is characterized by combination of exact and approximate methods.

In the phase of candidate generation, exact methods are employed in a-priori style. In the phase of IND checking, FAIDA uses several approximate approaches: Values in a column are processed either in probabilistic cardinality estimation or in a sampling-based method, according to the number of distinct values in the column.

The result of IND discovery by FAIDA may include not-INDs, but does not miss any IND; it may produce false positive but no false negative.

We referenced this overall framework of FAIDA in designing algorithms. Since our objective is designing accurate algorithms, we did not compare the performance with FAIDA.

7. Conclusion

In this paper we designed and implemented exact IND discovery algorithm applying acceleration methods; candidate generation which combines $(n-1)$-ary and 1-ary INDs, and IND checking with generating hash table with Position List Index.

We theoretically compared complexities of naïve methods and the method we propose. We also tested the performance in actual environment, and the result showed that our proposals are effective when processing relations above a certain scale.

We expect that further acceleration would be realized by incorporating other data profiling techniques, and also by applying parallel execution.

Acknowledgements

This work is partially supported by JST CREST Grant Numbers JPMJCR1414, JSPS KAKENHI Grant Number 17H01748, 19H04117 and project commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

References

[1] Abedjan, Z., Golab, L. and Naumann, F.: Profiling Relational Data: A Survey, *The VLDB Journal*, Vol. 24, No. 4, pp. 557-581 (2015).

[2] De Marchi, F., Lopes, S., and Petit, J. M.: Unary and n-ary inclusion dependency discovery in relational databases, *J. Intelligent Information Systems*, Vol. 32, No. 1, pp. 53-73, (2009).

[3] Heise, A., Quiané-Ruiz J., Abedjan, Z., Jentzsch, A., and Naumann, F.: Scalable Discovery of Unique Column Combinations, *Proceedings of the VLDB Endowment (PVLDB)*, Vol. 7, No. 4, pp. 301-312 (2013).

[4] Kruse, S., Papenbrock, T., Dullweber, C., Finke, M., Hegner, M., Zabel, M., Zöllner, C. and Naumann, F.: Fast Approximate Discovery of Inclusion Dependencies, *Datenbanksysteme für Business, Technologie und Web (BTW 2017),* pp. 207-226 (2017).

[5] Papenbrock, T., Bergmann, T., Finke, M., Zwiener, J., and Naumann, F.: Data profiling with metanome. *Proceedings of the VLDB Endowment (PVLDB)*, Vol. 8, No. 12, pp. 1860–1863. (2015) DOI:https://doi.org/10.14778/2824032.2824086