Parallelization of Inclusion Dependency Discovery and its Evaluation

Risa Ochiai† Kazuhiro Saito†‡ 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: †lisalisa@keio.jp, {ksaito, river}@sfc.keio.ac.jp †‡ku-saitou@kddi-research.jp

Abstract In recent years, data management tasks have become more complex due to the increasing variety of data. Such data management can be simplified by building metadata through data profiling. One of the data profiling tasks is inclusion dependency (IND) discovery which detects inclusion of data between columns. The problem with IND discovery is that the processing slows down because the amount of computation increases as the amount of data increases. In this paper, we propose a parallelization algorithm for IND discovery and report its high efficiency through evaluation.

KeyWord Data Profiling, Inclusion Dependency, Data Integration

1. Introduction

In recent years, a large amount of data has been generated and accumulated due to the increase in the use of the Internet and the spread of IoT. By collecting more data, one can perform higher quality data analysis and uses it for various purposes. However, as data diversity increases, data management becomes complex. On data management, one may explore desired data or try to discover relationships among a variety of data.

To deal with such problem, data profiling is effective. Data profiling [1] is a set of activities and processes to determine the metadata through a given dataset. Data profiling tasks can be divided into three categories: single-column, multiple-columns, and dependencies. Inclusion Dependency (IND) discovery is one of the profiling tasks to discover the inclusion of data between columns. IND indicates that all tuples of some columns in a relation are included in some other columns in the same or different relations [2]. IND is associated with several data management tasks such as foreign key detection and data integration, which are crucial for data profiling.

The purpose of this paper is to accelerate IND discovery, and we propose a parallel algorithm. IND discovery becomes complicated as the arity increases, and it takes long time to discover IND from a huge amount of data. By improving an existing IND discovery algorithm and parallelizing it, we present an efficient algorithm that is appropriate for handling a huge amount of data. The result of experiment shows the effectiveness of our proposed method.

The rest of this paper is organized as follows. In Section 2, we explain the general flow of multi-column IND discovery and the faster flow. In Section 3, we propose parallelization of IND discovery. In Section 4, we explain the evaluation method and shows the results. In Section 5, we describe the results. After explaining the related work in Section 6, we conclude this paper in Section 7.

2. Overview of IND discovery and Incremental IND discovery

This Section provides an overview of IND discovery. Let R be a relation, $|R|$ be the number of columns, and $|r|$ be the number of tuples. When selecting one or more columns from all the columns $r_1, r_2, \cdots, r_{|R|}$ in R, the selection method is determined as the combination R[a] of

the columns of R. Given the two relations R and S, if the tuple value of R[a] is included in the tuple value of S[b], there is a dependency of R[a] on S[b] and it can be expressed as R[a] ⊆ S[b]. The flow of candidate generation is described in Section 2.1.1, and the flow of IND discovery is described in Section 2.2.1. In addition, a flow speed up algorithm for candidate generation is described in Section 2.1.2, and a flow speed up algorithm for IND discovery is described in Section 2.2.2. In candidate generation, multiple columns can be selected from relations R and S respectively, a pair of column combinations that possible have dependency can be generated, and a pair referenced by an inclusive dependency can be generated. The IND check procedure scans the tuples to determine if the generated candidate is an IND relationship.

2.1 Candidate Generation Algorithm

Generating *n*-ary candidates is to select multiple (N) columns from relations R and S respectively, and it generates candidates that can be INDs.

2.1.1 Naïve Method

A naive method for the candidate generation simply take all the combinations of columns in each arity as an IND candidate. The algorithm for *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) Generate a Cartesian product on both sides and get IND candidates.

2.1.2 Fast Incremental Method

An efficient method referred to as fast incremental method prunes IND candidates which include a set of columns previously selected as a non-IND candidate [3]. In a unary case, the procedure is as follows. We combine (*n-1*)-ary and the unary IND to generate *n*-ary candidates.

(1) Select one column, each from R to generate a dependent side.
(2) Select one column each from S to generate a reference side.
(3) Generate a Cartesian product on both sides and get IND candidates.

In the case of *n*-ary, the procedure is as follows.

(1) Select n-1 columns from R to generate the dependent side.
(2) Select n-1 columns from S to generate a reference side.
(3) Select the furthest to the right column from R and multiply it by (1) and (2) respectively. At this time, each column on the referencing side must be unique.

This algorithm enables to remove candidates which cannot be INDs generated in naïve method and contribute to fast IND check phase.

2.2 IND Check Algorithm

The IND check algorithm is the process of determining if the generated candidates are really IND relationship.

2.2.1 Naïve Method

The naive algorithm to check IND is similar to the nested loops join algorithm. The algorithm checks all tuples on the reference side for each tuple on the dependent side.

2.2.2 Hashing with PLI Method

It is possible to accelerate this procedure by hashing the reference side as described in [3]. In this algorithm, each node in the hash table has a position list index (PLI) data structure. It contains the original tuple number in addition to the column values [4]. The reason for saving the tuple number is to determine if the same tuple contains values hashed in different hash tables when doing an IND check for *n*-ary.

3. Proposal

In the method explained in Section 2, the execution time of the IND discovery processing depends on the number of records multiplied by the number of candidates. This means

that the larger the size of target tables, the slower the processing. To solve this problem, we propose to accelerate this procedure by performing IND check by using multiple threads in parallel.

We propose the parallelization of the IND check phase based on the methods described in Section 2.1.2 and 2.2.2. The reason for not considering parallelization in the candidate generation phase is that the candidate generation method in Section 2.1.2 has a dependency between arities. Parallelization of task with dependencies is difficult as they say. In our proposed method, parallelization is performed by IND check for each arity, and PLI check of each candidate performed in each arity is executed in parallel by multiple threads without dependencies. Since this is embarrassingly parallel, appropriate efficiency is expected. When the number of cores is less than the number of candidates, the procedure is as follows.

Algorithm: Parallelization method

Input: R, S: relations
Output: INDlist: List of INDs from every arity

```
1    get the number of cores
2    if candidate count < number of cores then
3     for i=0 to candidate count do
4     create thread[i]
      start = i
      end = i + 1
5     IND check for ind candidates[start] to ind
     candidates[end] in thread[i]
6     end for
7    else if candidate count / number of cores = 0 then
8     for i = 0 to number of cores do
9     create thread[i]
10    start = candidate count / number of cores * i
11    end = candidate count / number of cores * (i+1)
12    IND check for ind candidates[start] to ind
     candidates[end] in thread[i]
13    end for
14   else
15    for i = 0 to number of cores do
16     if i=0then
17     start = 0
18     end = candidate count/number of cores
19     IND check for ind candidates[start] to ind
     candidates[end] in thread[i]
20     start = start + (candidate count + i − 1) /
     number of cores
21     end = start + (candidate count + i) / number of
     cores
22     else
23     IND check for ind candidates[start] to ind
     candidates[end] in thread[i]
24     end if
25    end for
26   end if
```

4. Evaluation

This Section compares conventional methods with our parallelization methods, and evaluates the performance of the IND discovery algorithm.

4.1 Implementation

We implemented the evaluation methods in C++ language. We used two CSV files when running the experiments. Each file contains one relation and checks if the data in the first file has an IND to the data in the second file.

4.1.1 Evaluation method

We experimented in an environment where the number of columns was changed and an environment where the number of records was changed. In the first environment, the number of records in the input file is fixed at 100 and the number of columns is changed. The second environment fixes the number of columns in the input file and changes the number of records.

The input file is a CSV file containing pseudo-random integer values. In the two experiments, specify the same csv file for the two files to be input, execute 5 times under the same conditions, and calculate the average of each execution time.

We explain the terms used in the figure. The upper limit of the number of columns in the csv file used in the experiment is the Limit of arity, and the upper limit of the number of records is the Limit of record. The execution time by the Candidate generation method in Section 2.1.1: naive method is Naive, and the execution time by the Candidate generation method in Section 2.1.2: fast incremental method is Fast incremental. The execution time by the IND check method in Section 2.2.1: naive method is Naive, and

the execution time by the Hashing with PLI method in Section 2.2.2 is PLI.

4.1.2 Experiment environment

We conducted experiments under the following conditions.

Hardware	Instance	AWS c5.9xlarge
	vCPU	36 cores
	Memory	72.00GiB
Software	OS	CentOS Linux release 7.7.1908 (Core) 64bit
	C compiler	g++ (GCC)7.3.1 20180303 (Red Hat 7.3.1-5)

4.2 Result

4.2.1 Evaluation on column scaling

We compared different candidate generation methods: the naive method and the fast incremental method.

In this experiment, the number of records in the input file was fixed at 100 and the number of columns was changed.

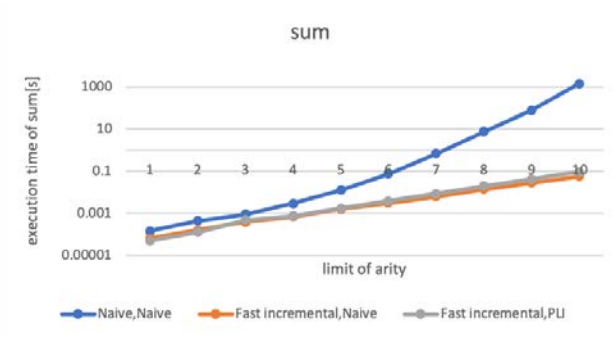

Figure 1

Comparison of averages of execution time of sum with different candidate generation methods

Fig. 1 shows the total of candidate generation execution time and IND check execution time of different candidate generation methods. From Fig. 1, it is observed that fast incremental method always shows faster execution time than naive method at 10 arities or less. Especially at 10 arities, the execution speed for the fast one was about 25,000 times faster than that of naïve one.

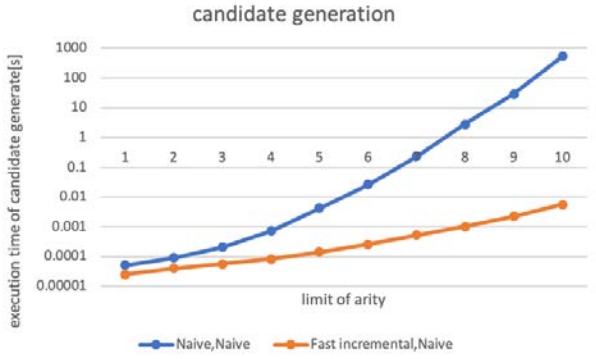

Figure 2

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

Fig. 2 compares the candidate generation execution time of different candidate generation methods. From Fig. 2, it can be observed that fast incremental method always has a faster execution time than naive method at 10 arity or less. Especially at 10 arity, the execution speed is about 100,000 times faster.

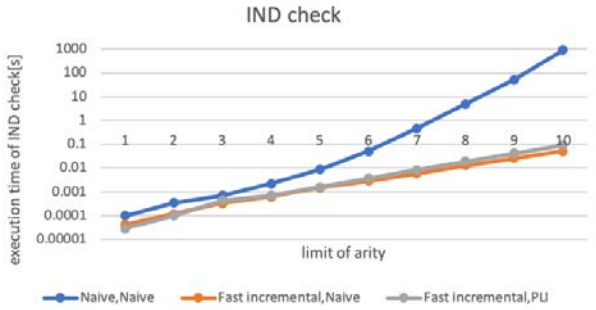

Figure 3

Comparison of averages of execution time of IND check with different candidate generation methods

Fig. 3 compares the IND check execution time of different candidate generation methods. From Fig. 3, it can be observed that fast incremental method is always faster than naive method at 10 arity or less as in the case of candidate generation. Especially at 10 arity, the execution speed was about 18,000 times faster. Comparing Naïve (orange) and Fast incremental, PLI (gray), It is observed

that Fast incremental, PLI (gray) are faster when the number of columns is 5 or more. The reason for this result is that hashing with PLI method does less processing by doing a hash search. Because the change in the number of columns doesn't change the amount of processing, it is necessary to change the number of records for analysis.

4.2.2 Evaluation on Record Scaling

Because changes in the number of records do not affect the performance of candidate generation, to measure the performance of IND check, we compared different IND checking methods: naive and hashing with PLI.

In this experiment, the number of columns in the input file was fixed at 6 and the number of records was changed. Candidate generation used fast incremental method. Limit of record indicates the upper limit of the number of records in the CSV file used in the experiment.

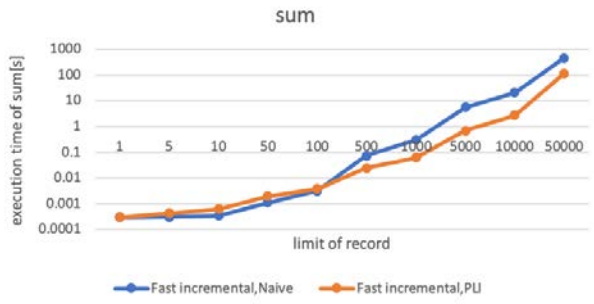

Figure 4

Comparison of averages of execution time of sum with different IND checking methods

Fig. 4 compares the sum of candidate generation execution time. In Fig. 4, the x-axis is the upper limit of record and the y-axis is the execution time [s], both of which are represented by logarithmic axes with a radix of 10.

It is observed that the execution speed of naive method is faster at 500 records or less, and hashing with PLI method is faster with more records. The execution speed of PLI was up to 4 times faster than that of naive.

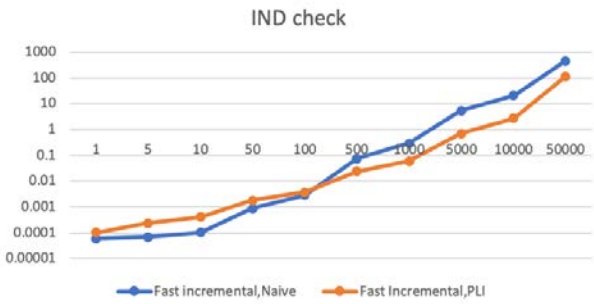

Figure 5

Comparison of averages of execution time of IND check with different IND checking methods

Fig. 5 compares IND check execution time for different IND check methods. In Fig. 5, the x-axis is the upper limit of record and the y-axis is the execution time [s], both of which are represented by logarithmic axes with a radix of 10.

It is observed that the execution speed of naive method is faster at 500 records or less, and hashing with PLI method is faster with more records, as in the sum result shown in Fig. 4. The execution speed for PLI was up to about 4 times faster than that of naive. From this result, it can be claimed that the number of records does not affect candidate generation time.

4.2.3 Evaluation of Parallelization

We evaluate the proposed method: parallelization of hashing with PLI method. We added a method of parallelizing IND check to the results in Section 4.2.2 for comparison.

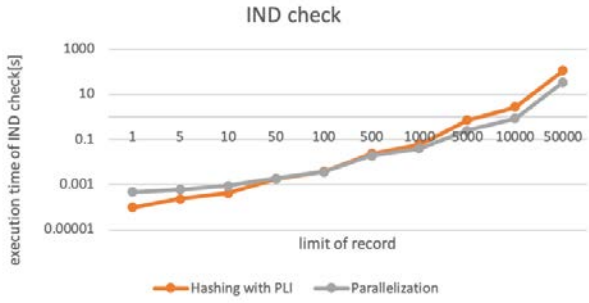

Figure 6

Comparison of averages of execution time of IND check with different IND checking methods

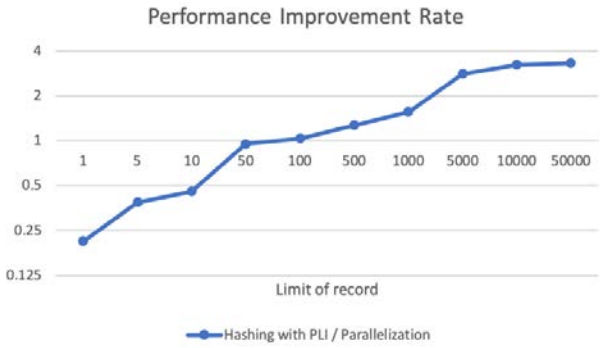

Figure 7

Performance improvement rate of IND check with different IND checking methods.

Fig. 6 compares IND check execution time of parallelization methods. Fig. 7 is performance improvement rate of IND check with parallelization method and hashing with PLI method. In Fig. 6, the x-axis is the upper limit of record and the y-axis is the execution time [s], both of which are represented by logarithmic axes with a radix of 10. In Fig. 7, the x-axis is the upper limit of record and the y-axis is the performance improvement rate, both of which are represented by logarithmic axes with a radix of 10.

From Fig. 6 and Fig. 7, it is observed that fast incremental, PLI(orange) is faster for 500 records or less, and Fast incremental, Parallelization(gray) is faster for 500 records or more. And, it is observed that the execution speed for the parallelization method is up to about 3.3 times faster that of serial method at 50,000 records.

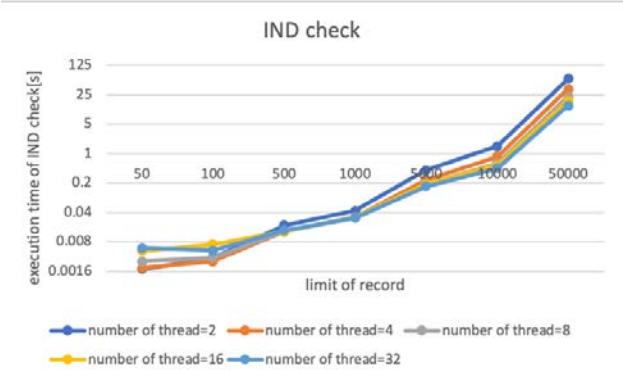

Figure 8

Comparison of averages of execution time of IND check with parallelization methods

Fig. 8 compares IND check execution time of parallelization methods. When we changed the number of thread in addition to change in the number of record, it was observed that the speed was increased as the number of threads increased with 1000 records or more. At 50,000 records, the execution speed of 32 threads is about 4.4 times faster than that of 2 threads.

5. Discussion

A comparison of candidate generation showed that fast incremental method was effective for all 10 arities and the difference was particularly large at 10 arities. This showed that fast incremental method efficiently generated candidates at least 10 arities or less, leading to fast and stable execution.

A comparison of IND checks showed that hashing with PLI method and parallelization method were effective for 500 records and above. It was considered that the result shown in Fig. 6 was obtained because the time required for thread creation was longer than the execution time of hashing with PLI method for 100 records or less. Also, comparing parallelization method and hashing with PLI method, the number of cores was set to 4 in this experiment, so it was expected to be four times faster, but actually, it was about three times faster.

A comparison of the number of threads shows that the speed is faster as the number of threads increases with 1000 records or more. It is thought that the reason why the speed is slower as the number of threads increases for 1000 records or less is that the number of candidates is small, so the effect of parallelization is not so great, and it is affected by the time it takes to create threads.

6. Related Work

IND discovery algorithm can be categorized to two classes. First class is unary IND discovery, and second class is n-ary

discovery. SPIDER [5] is a unary IND discovery algorithm based on adapted sort-merge join. This approach is known to be the most efficient algorithm in unary IND discovery algorithm. The algorithm has some limitation on application, and it cannot be applied to n-ary IND discovery. Our proposed method can be applied to n-ary cases. FAIDA [6] is an n-ary IND discovery algorithm that combines a probabilistic data structure with an approximate method. In the candidate generation phase, a-priori style candidates are generated. In the IND check phase, FAIDA uses an approximation technique to first detect all unary INDs and then iteratively generate and test IND candidates for each arity. As this is an approximate approach, the results of IND discovery by FAIDA may include non-IND.

7. Conclusion

In this paper, we designed a parallelization method for IND discovery and compared it with a method for accelerating IND discovery: an IND check that generates a hash table containing the position list index structure.

We compared naive method and proposed method: candidate generation method combining (n-1) -ary and n-ary and IND check method generating a hash table including position list index, both the candidate generation phase and the IND check phase are faster. In the candidate generation phase, Fast incremental method was always effective at 10 arity or less, and at 10 arity, it was about 100,000 times faster. As a result, it was found that candidates can be efficiently generated at 10 arity or less, and high-speed and stable execution can be performed.

In the IND check phase, Hashing with PLI method and the parallelization method were effective for 500 records or more. The parallelization method was up to about 3.3 times faster than the Hashing with PLI method with less than 50,000 records.

When the number of threads was changed, it was found that the speed was increased as the number of threads increased in 1000 records or more. At 50,000 records, the execution speed of 32 threads was about 4.4 times faster than that of 2 threads.

In future work, when we increase the number of threads, we plan to increase the amount of data used to the extent that there is a large difference in performance, and perform performance evaluation. Besides environment, we implemented parallelization for only the IND check phase in this work, but if parallelization of the candidate generation phase becomes possible, further speed up can be expected.

Acknowledgements

This work is partially supported by JP19H04117 and Project commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

Reference

[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] Hiroaki Uno, Kazuhiro Saito, Hideyuki Kawashima: An Acceleration of Inclusion Dependency Discovery and its Evaluation, *DEIM2020*, D8-3 (2020)

[4] 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).

[5] Bauckmann, J., Leser, U., Naumann, F., Tietz, V.: Efficiently detecting inclusion dependencies. In Proceedings of ICDE, pp. 1448–1450 (2007)

[6] 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).