Discovery of Unique Column Combinations with Hadoop

Shupeng Han, Xiangrui Cai, Chao Wang, Haiwei Zhang*, and Yanlong Wen

College of Computer and Control Engineering, Nankai University,
94 Weijin Road, Tianjin, P.R. China 300071
{hansp,caixr,wangc,zhanghw,wenyl}@dbis.nankai.edu.cn

Abstract. A unique column combination is one important kind of structural information in relations. From a data management perspective, discovering unique column combinations is a crucial step in understanding and utilizing the data. It will benefit data modeling, data integration, anomaly detection, query optimization and indexing. Nevertheless, discovering all unique column combinations is a NP-hard problem. Therefore, efficiency is a tremendous challenge.

In this paper, we propose *MRUCC*, which is an efficient algorithm to discover unique column combinations in large-scale data sets on Hadoop. Existing algorithms mainly focus on datasets of normal size, which cannot be adapted to large data sets. In contrast, we discover unique column combinations in parallel and implement *MRUCC* on Hadoop. Furthermore, we use column-based and row-based pruning to improve efficiency. Finally, we compare *MRUCC* with state-of-the-art approaches using both real and synthetic data sets. The experiment shows that *MRUCC* has a better performance.

Keywords: unique column combination, MRUCC, pruning, Hadoop.

1 Introduction

A unique column combination is one important kind of structural information in relations. A unique refers to a column or a column combination whose values uniquely identify a tuple in the collection. In other words, no two rows in a unique have identical values. Discovering uniques is a crucial step in many areas of modern data management, but the uniques are often incomplete in complex and large-scale data set. Since uniques represent a lot of valuable information, DBAs and developers are eager for an effective unique discovery method. However, it is impractical to manually detect all uniques in a large and complex data set. Thus, discovering uniques automatically and efficiently is very important in practice.

We focus on the discovery of non-redundant uniques instead of all unique column combinations. The non-redundant unique is a minimal column combination whose subsets are not uniques. It is worth noting that a column combination is

* Corresponding author.

a unique as long as one of its subsets is unique. Discovery of all unique combinations is especially difficult, because the combination number increases exponentially as attributes increase. *Gunopulos et al.*[1] pointed out that discovering a non-redundant unique is an NP-hard problem.

This paper proposes a parallel algorithm *MRUCC* (MapReduce based Unique Column Combination detection) for automatical discovery of non-redundant unique. There are two basic ideas behind *MRUCC*. We detect uniques that include the same number of columns in parallel. For example, we detect column combinations {A,B} and {A,C} simultaneously after verifying {A}, {B} and {C}. In addition, we use both column-based and row-based pruning to improve efficiency. Our pruning rules are: (1)if column combination {A} is a unique, any combination including {A} is a redundant unique; (2)if the count of identical tuples projected on {A} is 1, then we can ignore this tuple when verify the column combinations including {A}. We implement our algorithm on Hadoop. The experiments on both real and synthetic data sets show that *MRUCC* is better than other state-of-art approaches.

In general, we make three contributions.

1. To the best of our knowledge, we are the first to propose a parallel algorithm to discover unique column combinations on Hadoop. With the help of Hadoop, we can deal with large-scale data sets. Furthermore, our algorithm has a good scalability which is easily expanded to large-scale cluster.
2. We utilize column-based and row-based pruning techniques to accelerate our algorithm.
3. We demonstrate the efficiency of *MRUCC* on real and synthetic data sets.

The remainder of this paper is organized as follows. In section 2, We review the related work. Then we discuss two important pruning techniques in section 3. Section 4 contains the description of the *MRUCC* algorithm. Section 5 shows the results of an empirical evaluation of *MRUCC*. At last, we conclude this paper in section 6.

2 Related Work

Currently, only a few of research work aimed to solve unique discovery problem for its complexity. [2,3,4,5] are devoted to detecting uniques of single column, instead of composite uniques. Most existing work can only be used for small-scale data sets due to the limitation of CPU and memory capacity. In general, there are two different kinds of solutions: column-based and row-based.

BruteForce is a column-based algorithm. It needs to traverse all column combinations of a relational table, and verifies each of them by the method of *'projection and count'*. Then it removes redundant combinations and eventually gets all non-redundant uniques. In conclusion, the *BruteForce* algorithm consists of two phases: a unique verification phase and a redundancy removal phase. Assuming that a data table has 50 attributes, thus there is $2^{50} - 1$ column combinations altogether to be verified. Therefore, *BruteForce* can only be used for small-scale data set with a few columns.

The typical row-based algorithm is *GORDIAN*[6], which is based on the intuition that non-uniques can be detected without traversing all rows in a table. *GORDIAN* firstly reorganizes the data table as a prefix-tree to fit into memory, and computes maximal non-uniques by traversing the prefix-tree. Then, it computes minimal uniques from maximal non-uniques. However, it needs the prefix-tree to be fully loaded into memory, which is difficult when the data set is very large. What's more, generating minimal uniques from maximal non-uniques is also the bottleneck of *GORDIAN*.

HCA[7] is an algorithm based on bottom-up apriori technique. *HCA* uses an efficient candidate generation strategy and applies statistical pruning with value histograms. Besides, it takes advantage of functional dependencies and combines the maximal non-unique discovery with *HCA*. However, there is no optimization with regard to early identification of non-uniques in a row-based manner as *HCA* is based on histograms and value-counting.

3 Pruning Methods of Unique Discovery

In section 3, we introduce two important pruning methods used in *MRUCC*, which improve the efficiency of unique discovery. Column-based pruning is designed to prune unique candidates. Row-based pruning can be used to accelerate the verification of a column combination.

In order to verify a column combination, we need to project the data table onto this combination and get a new table S. It proves that combination K is a unique only if there is no duplicate tuples in S.

3.1 Basic Concepts

Unique column combinations are sets of columns of a relational database that fulfill the uniqueness constraint. Uniqueness of a column combination K within a table can be defined as follows[7]:

Definition 1. Unique *Given a relational database schema* $R = \{C_1, C_2, ..., C_m\}$ *with columns* C_i *and instance* $r \subseteq C_1 \times ...C_m$*, a column combination* $K \subseteq R$ *is a unique iff* $\forall t_1, t_2 \in r : (t_1 \neq t_2) \Rightarrow (t_1[K] \neq t_2[K])$*.*

Minimality is an important property of unique. Minimal uniques are uniques of which no strict subsets hold the property.

Definition 2. Minimal Unique *An unique* $K \subseteq R$ *is minimal,* iff $\forall K' \subset K : (\exists t1, t2 \in R : (t1[K'] = t2[K']) \wedge (t1 \neq t2))$*.*

3.2 Column-Based Pruning

If both K and K' are uniques of schema R, and $K \subset K'$, then we remark K' as the redundancy of K. As shown in Figure 1, a schema R has 4 columns a, b, c, d and its uniques are $\{b\}$ and $\{a, c\}$. If we use *BruteForce* algorithm to find all non-redundant uniques, we need to verify all combinations in the column combination

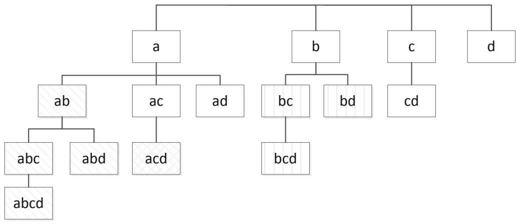

Fig. 1. Column combination tree(CC-Tree)

tree(CC-Tree for short), which has 15 candidates altogether. Heuristically, we can apply column-based pruning to avoid unnecessary verification. For example, we should check every column combination showed in figure 1 in breadth-first order. After confirming that $\{b\}$ is a unique, we can omit the verification of $\{b, c\}$, $\{b, d\}$ and $\{b, c, d\}$ since that they are all redundancies of $\{b\}$. With column-based pruning, it only takes seven verification to find all non-redundant uniques rather than 15 times. The benefits of column-based pruning get more considerable when the table has more attributes.

3.3 Row-Based Pruning

Row-based pruning is used to shrink computational cost of verifying a column combination. Recall that we verify candidates in bread-first order(Figure 1). If a tuple appears only once in the projection of a column combination K, we can ignore this tuple when verify the subtree of K. For example, when we verify column combination {Major} showed in Table 1, we can obtain that {Major} is not a unique. Next it needs to verify combinations that are constituted by {Major} and another attribute, namely {Major,Name}, {Major,Grade} and {Major,StudentID}.

Table 1. Data set sample

Name	Major	Grade	StudentID
Lucy	Computer	freshman	1210368
John	Math	freshman	1210422
Lucy	English	Junior	1010232
Lucy	Computer	Sophomore	1110223

There is only one record whose value is *Math* in the projection on {Major}, so there must be only one record whose *Major* is *Math* in the projection on {Major,Name}, {Major,Grade} and {Major,StudentID}. It is obvious that the second tuple will not affect the verification of these three column combinations, and the same as the third tuple. Therefore, we can prune the data table by removing the tuples that appear only once in the projection and we get a and smaller table shown in Table 2.

Table 2. Pruned data set

Name	Major	Grade	StudentID
Lucy	Computer	freshman	1210368
Lucy	Computer	Sophomore	1110223

4 MRUCC Algorithm

In this section, we introduce our unique discovery algorithm *MRUCC*. It is proposed to deal with large-scale data set on Hadoop with column-based and row-based pruning.

The core of *MRUCC* algorithm is to verify all column combinations located in the same layer of the CC-tree simultaneously. Assume that *MRUCC* is to verify the column combination set of $\{C_1, C_2, ... C_n\}$. Firstly it traverses this set and checks redundancy for each combination C_i. If C_i is not redundant, *MRUCC* adds C_i into a set named *CandidateList*, which is used to maintain combinations that need further verification. Otherwise, if C_i is a redundant unique, all the descendant nodes of C_i will be pruned. After traversing, all combinations in *CandidateList* will be verified simultaneously. If C_i is a unique, *MRUCC* would add C_i into *Unique-Set* and prune its subtree. Then, *MRUCC* processes next level of the CC-Tree.

The *MRUCC* algorithm(Algorithm 1) takes *ContextInfoList* as its input. *ContextInfoList* is a list constituted by *ContextInfo* objects, which is a data structure used to build candidate column combinations. *MRCheck* is a unique verification algorithm that contains a *Mapper* class and a *Reducer* class. The former is responsible for the projection of each column combination, and the latter is in charge of aggregation. We materialize the new table after row-based pruning, in order to accelerate the verification of subtree.

Consider a data table with four attributes a, b, c and d, where {b} and {a,c} constitute its *Unique-Set*. The corresponding CC-Tree is shown in Figure 1. The *MRUCC* algorithm first finds uniques from the first layer of CC-Tree: it checks redundancy of {a}, {b}, {c}, {d} and finds that all of them are not redundant, so it sends these four candidates to *MRCheck* for unique verification. Then it prunes the subtree of {b} since candidate {b} is a unique. Next, the algorithm is processed to the second layer of the CC-Tree, where column combinations are {a,b}, {a,c}, {a,d}, {c,d}. Combination {b,c} and {b,d} are not be processed because they have been pruned before. Since {a,b} is redundant for {b}, the *MRUCC* prunes its subtree analogously. It then passes {a,c}, {a,d}, {c,d} to *MRUCC* and finds that {a,c} is a key, so it performs pruning similarly. At last, the algorithm is processed to the third layer of the CC-tree and it comes to the end since no combinations left in this layer.

Algorithm 1. MRUCC

```
Input: ContextInfoList
Output: Unique collection UniqueSet
1: if ContextInfoList is null then
2:    return UniqueSet
3: end if
4: for each info in ContextInfoList do
5:    Construct Candidate using info and insert it into CandidateList;
6: end for
7: if CandidateList is not null then
8:    Invoke MRCheck(CandidateList) for unique verification concurrently
9: end if
10: clear ContextInfoList;
11: for all Candidate ∈ CandidateList do
12:    if Candidate is a unique then
13:       Insert Candidate into UniqueSet
14:    else
15:       Construct new ContextInfo node and insert it into ContextInfoList
16:    end if
17: end for
18: invoke MRUCC(ContextInfoList)
```

5 Experiment

We use a series of experiments to demonstrate that *MRUCC* has better performance than the state-of-art algorithms on large scale data set. In addition, it also proves that *MRUCC* algorithm has a good scalability.

5.1 Setup

Cluster: Our Hadoop environment has totally 6 nodes: one master and 5 slaves.

Data Sets: We use one synthetic and two real-world data sets in our experiments. TPC-E is a data set which is used to simulate the OLTP workload of a brokerage firm. *FinancialData* and *MovieLens* are real-word data sets. The former is from the economic field while the latter is from the film industry. The statistics of each data set are shown in Table 3.

Table 3. Statistics of data sets

Data Set	Table	Avg Col	Max Col	Avg Row	Max Row
TPC-E	32	6	24	19285770	207407310
FinancialData	8	7	16	134288	1056320
MovieLens	12	6	15	130084	855599

5.2 Scaling the Number of Columns

In this section we run *MRUCC* and *GORDIAN* on data sets with different column numbers to compare their run time with the same computer. To thoroughly evaluate their performances, different column numbers and different data set sizes are utilized. The experimental results are reported in Figure 2, Figure 3 and Figure 4 respectively. The cost of *GORDIAN* increases exponentially with the growth of column number, while the curve of *MRUCC* is relatively flat. In addition, *MRUCC* has a better performance when the data set gets lager, which proves that *MRUCC* is efficient.

5.3 Scaling the Number of Rows

To analyze the effect of row number, we run *MRUCC*, *BruteForce* and *GORDIAN* on data sets with different size on a single computer. Figure 5 shows the execution time of these three algorithms.

As can be seen from Figure 5 that the cost of the *MRUCC* increases nearly in a linear fashion as the data set size grows. Besides, *MRUCC* is much better than *BruteForce* all the time. When the data size is small, *GORDIAN* has a better performance than *MRUCC*, since the parallel overhead account for a large percentage; but when the size gets larger, *MRUCC* is superior to *GORDIAN*. In conclusion, *MRUCC* algorithm is more suitable for dealing with large data sets.

5.4 Speed-Up Ratio of Cluster

Speedup ratio is an important criterion to measure the performance of parallel algorithms. The speedup ratio is defined as $S = T_s/T_p$, where T_s is the serialize execution time and T_p denotes the run time of parallel algorithm with p nodes. In this section, we randomly select 1/64, 1/32, 1/16, 1/8, 1/4, 1/2, and the entirety of a 6GB data table as the sample sets. Then we evaluate speed-up ratio with these sets on different number of nodes. As shown in Figure 6, the *MRUCC* algorithm increases in a linear fashion when the data set is large, which proves that *MRUCC* has a good speedup ratio.

5.5 Expansion Rate of Cluster

Expansion rate, which is defined as $E = S/P$, is used to assess the utilization of cluster. S is the speedup ratio, while P denotes node number. Figure 7 shows that *MRUCC* has a good scalability. The expansion rate is gradually decreased with the growth of node number, since the cost of communication between nodes increases. As the number of nodes grows, the expansion rate curve becomes smooth gradually. It can give full play to its computing ability for each node when the data size is large.

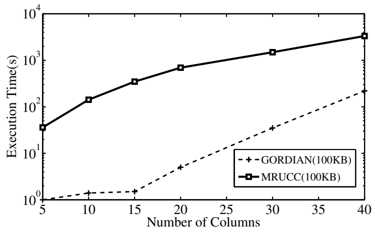

Fig. 2. Column number effect(100KB)

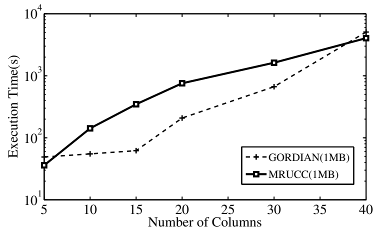

Fig. 3. Column number effect(1MB)

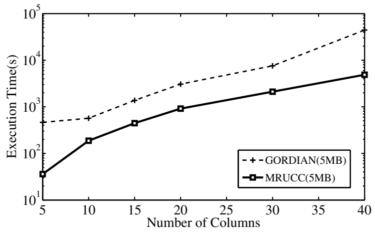

Fig. 4. Column number effect(5MB)

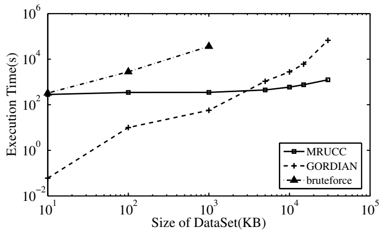

Fig. 5. Row number effect

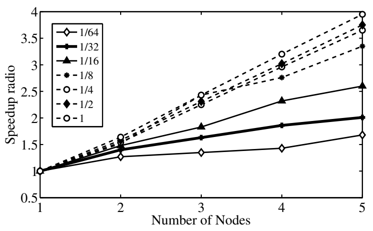

Fig. 6. Speed-up ratio

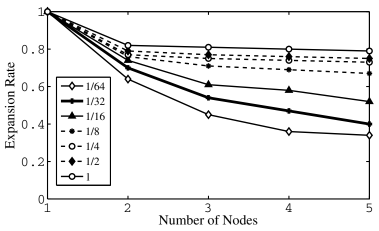

Fig. 7. Expansion rate

6 Conclusion

A unique is the basis of understanding a data table, but it is often incomplete in large data set. This paper proposes a distributed non-redundant unique discovery method named *MRUCC*, which is based on Hadoop platform. Additionally, We use column-based as well as row-based pruning to improve its efficiency.

There are some works to be done in future. Firstly, we can cache intermediate results of *MRUCC* to avoid unnecessary I/O cost. Furthermore, we can apply data partitioning to find as many non-uniques as possible on these subsets, so as to improve column-based pruning.

Acknowledgments. This work is supported by NSF of China under Grant No. 61170184, and Tianjin Municipal Science and Technology Commission under Grant No. 13ZCZDGX02200, 13ZCZDGX01098, 13JCQNJC0100 and 14JC-QNJC00200.

References

1. Gunopulos, D., Khardon, R., Mannila, H., Saluja, S., Toivonen, H., Sharma, R.S.: Discovering all most specific sentences. ACM Trans. Database Syst. 28(2), 140–174 (2003)
2. Brown, P., Haas, P.J., Myllymaki, J., Pirahesh, H., Reinwald, B., Sismanis, Y.: Toward automated large-scale information integration and discovery. In: Härder, T., Lehner, W., et al. (eds.) Data Management in a Connected World. LNCS, vol. 3551, pp. 161–180. Springer, Heidelberg (2005)
3. Bell, S., Brockhausen, P.: Discovery of constraints and data dependencies in databases. In: Lavrač, N., Wrobel, S. (eds.) ECML 1995. LNCS, vol. 912, pp. 267–270. Springer, Heidelberg (1995)
4. Kivinen, J., Mannila, H.: Approximate dependency inference from relations. Theoret. Comput. Sci. 149, 129–149 (1995)
5. Petit, J.-M., Toumani, F., Boulicaut, J.-F., Kouloumdjian, J.: Towards the reverse engineering of renormalized relational databases. In: Proc. ICDE, pp. 218–227 (1996)
6. Sismanis, Y., et al.: GORDIAN: efficient and scalable discovery of composite keys. In: Proceedings of the 32nd International Conference on Very Large Data Bases. VLDB Endowment (2006)
7. Abedjan, Z., Naumann, F.: Advancing the discovery of unique column combinations. In: Proceedings of the 20th ACM International Conference on Information and Knowledge Management. ACM (2011)
8. Adelfio, M.D., Samet, H.: Schema extraction for tabular data on the web. Proceedings of the VLDB Endowment 6(6), 421–432 (2013)
9. Janga, P., Davis, K.C.: Schema extraction and integration of heterogeneous XML document collections. In: Cuzzocrea, A., Maabout, S. (eds.) MEDI 2013. LNCS, vol. 8216, pp. 176–187. Springer, Heidelberg (2013)