Smart Caching for Efficient Functional Dependency Discovery

Anastasia Birillo[1,2] and Nikita Bobrov[1,2(✉)]

[1] JetBrains Research, Saint-Petersburg, Russia
anastasia.i.birillo@gmail.com, nikita.v.bobrov@gmail.com
[2] Saint Petersburg State University, Saint-Petersburg, Russia

Abstract. Functional dependency (FD) is a concept for describing regularities in data, that traditionally was used for schema normalization. Recently, a different problem has emerged: for a given dataset, find FDs that are contained in it. Efficient FD discovery is of great importance due to FD usage in many tasks such as data cleaning, schema recovery, and query optimization.

In this paper we consider a particular class of FD discovery algorithms that rely on partition intersection. We present a simple approach for caching intermediate results that are generated during run times of these algorithms. Our approach is essentially a heuristic that selects which partitions to cache based on measures of disorder in data. For this purpose, we adopt such well-known measures as Entropy and Gini Impurity, and propose a novel one—Inverted Entropy. Our approach has a negligible computational overhead, and can be used with a number of FD discovery algorithms, both exact and approximate. Our experiments demonstrate that our heuristic allows to decrease algorithm run times by up to 40% while simultaneously requiring up to an order of magnitude less space compared to the state-of-the-art caching approaches.

Keywords: Functional dependency · Partition intersection · Partition caching · Caching · Entropy · Gini impurity · PYRO · TANE

1 Introduction

Functional dependency (FD) states that some attributes of a table functionally determine the value of some other attribute, or a set of other attributes. Since their introduction by Codd [2] in 1969, they have been widely applied, mainly for schema normalization, i.e. in this case they were known in advance.

However, in the 90's the focus of research community shifted to different scenarios, where one has to discover FDs for a given dataset. This problem

This is a student work, authored solely by students. Advisors: George Chernishev (chernishev@gmail.com) and Kirill Smirnov (kirill.k.smirnov@gmail.com). This work is partially supported by RFBR grant 16-57-48001.

T. Welzer et al. (Eds.): ADBIS 2019, CCIS 1064, pp. 52–59, 2019.
https://doi.org/10.1007/978-3-030-30278-8_7

formulation gained even more popularity with the advent of big data era, because the task of data exploration became much more common.

The problem of FD discovery is very computationally demanding, since the number of potential irreducible FDs grows as $O(N * 2^N)$, where N is the number of involved attributes. Moreover, algorithm run times depend not only on the number of attributes, but on the number of rows as well. In the 90's, the state-of-the-art algorithms on the average desktop PC could handle [4] datasets of up to 20 attributes and tens of thousands of rows within reasonable time (up to several hours). Nowadays [7,8], contemporary algorithms run on a server-class many-core multiprocessor computer discover dependencies for datasets consisting of hundreds of attributes (up to 200) and tens of thousands of rows (up to 200K) within roughly the same time limits.

No doubt, this is substantial progress. Nevertheless, it is insufficient: these volumes are unacceptable for the majority of real-life applications. Therefore, efficient FD mining remains an acute problem that requires designing new and speeding up existing algorithms.

Many algorithms for FD discovery (including some of the recent ones, e.g. [5]) rely on the notion of partition. Informally, a partition for a given attribute X is a collection of lists where each list contains row numbers that have the same value for X. Next, this notion can be straightforwardly generalized for an arbitrary number of attributes. The core of algorithms that employ this concept is the operation of partition intersection. It is rather costly and more importantly, it is performed frequently. However, it can greatly benefit from caching since the algorithm may intersect the same partitions multiple times. Therefore, several existing algorithms use partition caching.

In this paper we propose a novel caching mechanism that employs a heuristic to guide cache population. Our idea is the following: during partition intersection, check the heuristic for the result and if it passes, add it to the cache. This heuristic is based on a measure of "uniqueness" of values of an attribute set. For this purpose, we have adopted several existing measures such as Entropy, Gini Impurity, and developed a new one, which we named Inverted Entropy. Our technique has a negligible computational overhead and can be used with a number of FD discovery algorithms, both exact [4,9] and approximate [5]. We have also proposed a novel cache eviction policy, that captures usefullness of partitions and tries to retain the most useful ones.

Evaluation performed on a number of real datasets has demonstrated that our caching approach allows to improve run times up to 40%, while simultaneously decreasing the required memory by up to an order of magnitude.

2 Background: Partitions and Caching

Many existing FD discovery algorithms employ the partition intersection approach [7]. Formally, a *stripped partition* or *position list index* (PLI) is defined as follows [5] (for the ease of comprehension, we will follow the Kruse and Naumann notation throughout our paper):

Definition 1. *Let r be a relation with schema R, and let $X \subseteq R$ be a set of attributes. A cluster is a set of all tuple indices in r that have the same value for X, i.e., $c(t) = \{i | t_i[X] = t[X]\}$. The PLI of X is the set of all such clusters except for singleton clusters:*

$$\overline{\pi}(X) := \{c(t) | t \in r \wedge |c(t)| > 1\}$$

Next, the *size* of a PLI is defined as $\|\overline{\pi}(X)\| = \sum_{c \in \overline{\pi}(X)} |c|$. Essentially, it is the number of included tuple indices. Finally, *arity* of a partition is defined as the number of involved attributes and we *atomic partition* a partition that has arity of 1.

PLIs are extensively used for FD validation, i.e. for checking whether a given FD holds. A frequently used criterion [4] is the following: $X \rightarrow A$ $A, X \subseteq R$ holds $\Leftrightarrow \|\overline{\pi}(X)\| = \|\overline{\pi}(X \cup A)\|$. To compute $\overline{\pi}(X \cup A)$, a procedure called PLI intersection (also known as partition product) is performed. Its idea is to avoid the costly rescan of the necessary attributes for computing $\overline{\pi}(X \cup A)$ via reusing the already known $\overline{\pi}(X)$ and $\overline{\pi}(A)$. This procedure significantly contributes to algorithm run time, since it is invoked many times during the search space traversal. However, it is often called for overlapping attribute sets. Therefore, PLIs are promising for caching.

Several recent FD discovery algorithms employ caching [1,5]. In study [5] authors use a set-trie data structure, which is essentially a prefix tree where each node stores the corresponding PLIs. In this paper, the authors have proposed a sophisticated algorithm for cache lookup, but they relied on coin flip for cache population (see algorithm 2). In our study, we argue that such an important part should not be left to chance.

3 Our Approach

3.1 Preliminaries

Consider the following. Let $D = X \rightarrow Y$ be an arbitrary FD, and for simplicity, let X and Y be single attributes. The following observations are true:

1. If X contains unique values, then D holds regardless of Y.
2. If Y contains unique values, then D holds only if X contains only unique values too.
3. If X contains equal values, then D holds only if contains Y equal values too.
4. If X contains equal values, but Y has at least one non-equal value, then D is violated.

Therefore, the more "unique" values are in X, the more probable that D will hold. For Y it is vice versa: the more similar values are in Y, the more probability that the dependency will hold.

Next, note that partition-based FD discovery algorithms share a common pattern: when a dependency is violated, they try to fix it via introducing additional attributes to the left part (in some papers, this procedure is called specialization [4]).

Our idea is the following: PLIs obtained during the course of the algorithm are not equally worthy of caching. There is no need to cache PLIs for dependencies that are almost satisfied, i.e. that have a high probability of being satisfied during the next specialization. Contrary to this, PLIs that would require many rounds of specialization should be cached.

As shown above, this probability is directly linked to the degree of "uniqueness" of values of dependency's left hand side. Note that this probability is a monotonic, non-decreasing function with regard to specialization steps. In other words, after a specialization step, the probability that this new (specialized) dependency holds is at least the same as before.

There are several ways to calculate the degree of "uniqueness" of a set of values:

1. Shannon Entropy is a classic measure of disorder in data. It is given as follows:

$$H = -\sum_i P_i \log P_i,$$

   where P_i is the probability mass function. The entropy is 0 if there is a single value for all entries for a given attribute. If all records contain different values for this attribute (e.g. the attribute is the primary key) then entropy is $\log n$, where n is the number of records.
2. Gini impurity is a measure used for construction of decision trees. It is computed as follows:

$$I_G = 1 - \sum_{i=1}^{J} P_i^2,$$

   where J is the number of classes and P_i is the probability of correctly classifying i-th item.
3. We have also proposed our metric, which is essentially a minor modification of Shannon Entropy:

$$H^{-1} = -\sum_i (1 - P_i) \log(1 - P_i).$$

   Such a metric inherits the behaviour of the previous ones (it also grows as a set of values becomes more unique), and its sensitivity to "uniqueness" can be compared to the other metrics as follows: $I_G \leq H^{-1} \leq H$.

3.2 Caching Algorithm and Cache Population

In order to evaluate our ideas, we have decided to modify the PYRO algorithm [5] by introducing our heuristic. In their paper, cache lookup and cache population were contained in a single algorithm as described below. Suppose that we have to compute the result of partition intersection $\overline{\pi}(X)$, where X is an attribute set.

1. If the required $\overline{\pi}(X)$ is not present in cache, try to compute it by efficiently reusing cached results for subsets of X. The idea is to minimize the number of intersections and to keep the size of intermediate results small.
2. While incrementally constructing $\overline{\pi}(X)$, for each intermediate result $\overline{\pi}(C)$ decide whether to cache it or not. The decision is made randomly, depending on the outcome of the coin toss.

We propose to use the following heuristic to guide caching:

$$if(PLIUniqueness(C) \leq medianUniqueness * (1 + mod_1(C) + mod_2)) \; then \; cache.$$

Here, $PLIUniqueness(C)$ and $medianUniqueness$ are "uniqueness" degrees of newly computed PLI and the median of "uniqueness" degrees for individual attributes. In our study we experimentally evaluate all three "uniqueness" measures that were described above.

The modifiers are as follows:

$$mod_1(C) = \frac{\max_{c \in \overline{\pi}(C)} |c|}{N}, \; mod_2 = \frac{maxHeapSize - availableMemory}{maxHeapSize}.$$

Here, $|c|$ is cluster size and N is relation size, both in records. The mod_1 modifier indicates how close our newly computed PLI is to being primary key. It is $1/N$ when it is primary key (all values are unique) and 1 when its values are all the same. This way, we stimulate caching of PLIs that are far from being primary key, because dependencies with such left hand sides are likely to fail.

In the second modifier, $availableMemory$ and $maxHeapSize$ represent currently available and maximum available memory respectively (in megabytes). This modifier allows to keep track of cache occupancy and allows to favour population while there is plenty of free space.

It is necessary to note that the proposed heuristic is computationally lightweight: everything that is required can be computed during partition intersection.

3.3 Cache Eviction

During the course of the algorithm there is always a chance to run out of available memory, making it impossible to add more PLIs into the cache. The authors of PYRO suggest to perform cache eviction (shrink procedure) each time when the cache size is more than $modifer * maxHeapSize$ ($modifier$ set to 0.85 by default setting). The original procedure is performed as follows: (1) construct a priority queue based on arity of all cached PLIs, starting from PLIs with the arity of 2; (2) remove PLIs from the cache until target size requirement of the cache is met (half of the original size by default). Obviously, such an approach does not take into consideration the real usefulness of PLIs.

We propose a PLI eviction strategy that is based on the number of PLI accesses. This statistic in combination with the cache population strategy

described above can help to collect and retain useful PLIs. Therefore, we modified the original eviction algorithm as follows: (1) every time a new PLI is constructed, increment the number of accesses for each cached PLI that was used for its construction; (2) when the shrinking is run, calculate the median of a PLI's number of accesses; (3) add all PLIs that have a less than median number of accesses into a priority queue; (4) clean cache by polling the queue and disposing elements on a per PLI basis until the target size of the cache is met; (5) set the number of accesses of remaining PLIs to zero.

4 Experiments

We have implemented our approach in the Metanome [6] system and used the PYRO algorithm (with agree-set sample size parameter of 10,000) for comparison. In our experiments, we have used the following hardware and software configuration: Intel(R) Core(TM) i5-4670K CPU 3.40GHz (4 CPUs) RAM 16GB, Ubuntu Linux 18.04 (64 bit). The characteristics of the used tables are presented in the left part of Table 1.

We have conducted two experiments. In Fig. 1 we present the results of the run time experiment with our cache population approach and PYRO's eviction policy. We can see that heuristic cache population always improves run times compared to the default approach (Coin). Depending on the dataset, improvement ranges from 10% to 40%. However, there is no clearly superior data disorder measure. In this experiment we have also measured the volume of data which was put into cache (presented in Table 1, right part) during the whole run time, including evictions. We can see that our heuristic allows to drastically reduce the amount of cached data, sometimes up to an order of magnitude.

The second experiment assessed our cache population approach and our eviction policy. To force frequent invocations, we have set a tight memory limit of 512 MBs. Due to that, run times increased by almost two times. We do not present a bar chart or table with figures due to space constraints, but in this experiment heuristic population approach was also superior to the coin-based one in terms of run times. This experiment also did not identify the best measure, i.e. some measures excel on different tables, but all of them are superior to baseline.

5 Related Work

There are several dozen of algorithms for FD discovery. We will not survey all of them due to space constraints, instead referring an interested reader to the study [7].

Most of these algorithms employ partition intersection and some of those can benefit from partition caching. However, to the best of our knowledge, there are only two algorithms that implement such caching techniques.

The first one is PYRO [5], where caching is performed randomly (via coin toss) for every partition that was computed during intersection.

Table 1. Used tables, their details and memory consumption (MB) of different caching approaches.

Table	Records	Attrs	#FD	Coin	Entropy	Gini	Inverted entropy
ditag	3,960,124	13	171	1363	665	681	649
iowa	27,373,453	24	1202	15625	9281	5430	7656
structSheet	664,128	32	1058	837	180	1,267	13
pdbx	29,787	35	108001	71,503	59,103	21,302	35,340
apogeeStar[a]	277,370	49	71066	170,126	165,578	73,351	41,553
plista	1,000	63	316866	10,312	1435	4512	570
flight	1,000	109	144185	1,220	409	125	943

[a]Sloan Digital Sky Survey. https://www.sdss.org/dr14/

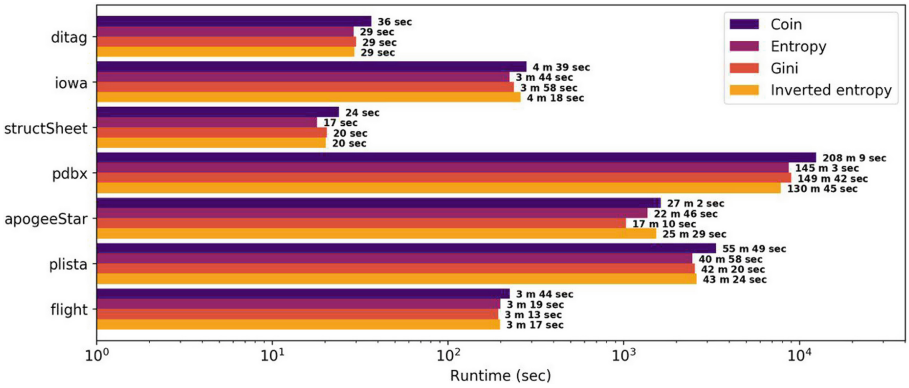

Fig. 1. Runtime comparison of different caching approaches.

The other one is DFD [1] that has adopted caching from the DUCC [3] algorithm. Similarly to PYRO, its cache population strategy is also straightforward: the authors propose to cache all obtained partitions. However, their cache eviction strategy is more sophisticated. First, during the course of the algorithm, calculate the number of uses for each partition. Second, whenever the number of stored partitions exceeds a threshold, run cache clean-up. During this process, remove all non-atomic partitions whose use counts are below the median value of the currently stored partitions.

6 Conclusion

In this paper, we have addressed the problem of intermediate result caching for partition-based FD discovery algorithms. We have proposed to use three measures of disorder in data to guide the cache population algorithm. Next, we have designed a simple cache eviction algorithm to conserve memory. To experimentally validate our approach, we have implemented it inside the Metanome system and modified its PYRO. Experimental results show that our approach decreases

run times up to 40% on a number of datasets, while simultaneously decreasing the volume of used memory by up to an order of magnitude. The proposed approach has a negligible computational overhead and, more importantly, can be used with a different partition-based FD discovery algorithms such as TANE.

References

1. Abedjan, Z., Schulze, P., Naumann, F.: DFD: efficient functional dependency discovery. In: Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, pp. 949–958. ACM, New York (2014)
2. Codd, E.F.: Further normalization of the data base relational model. IBM Research Report, San Jose, California, RJ909 (1971)
3. Heise, A., Quiané-Ruiz, J.-A., Abedjan, Z., Jentzsch, A., Naumann, F.: Scalable discovery of unique column combinations. Proc. VLDB Endow. **7**(4), 301–312 (2013)
4. Huhtala, Y., Kärkkäinen, J., Porkka, P., Toivonen, H.: TANE: an efficient algorithm for discovering functional and approximate dependencies. Comput. J. **42**, 100–111 (1992)
5. Kruse, S., Naumann, F.: Efficient discovery of approximate dependencies. Proc. VLDB Endow. **11**(7), 759–772 (2018)
6. Papenbrock, T., Bergmann, T., Finke, M., Zwiener, J., Naumann, F.: Data profiling with metanome. Proc. VLDB Endow. **8**(12), 1860–1863 (2015)
7. Papenbrock, T., et al.: Functional dependency discovery: an experimental evaluation of seven algorithms. Proc. VLDB Endow. **8**(10), 1082–1093 (2015)
8. Papenbrock, T., Naumann, F.: A hybrid approach to functional dependency discovery. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD 2016, pp. 821–833. ACM, New York (2016)
9. Wang, S.-L., Shen, J.-W., Hong, T.-P.: Incremental discovery of functional dependencies using partitions. In: Proceedings Joint 9th IFSA World Congress and 20th NAFIPS International Conference (Cat. No. 01TH8569), vol. 3, pp. 1322–1326, July 2001