Advancing the Discovery of Unique Column Combinations*

Ziawasch Abedjan
Hasso-Plattner-Institut
Potsdam, Germany
ziawasch.abedjan@hpi.uni-potsdam.de

Felix Naumann
Hasso-Plattner-Institut
Potsdam, Germany
naumann@hpi.uni-potsdam.de

ABSTRACT

Unique column combinations of a relational database table are sets of columns that contain only unique values. Discovering such combinations is a fundamental research problem and has many different data management and knowledge discovery applications. Existing discovery algorithms are either brute force or have a high memory load and can thus be applied only to small datasets or samples. In this paper, the well-known GORDIAN algorithm [9] and "Apriori-based" algorithms [4] are compared and analyzed for further optimization. We greatly improve the Apriori algorithms through efficient candidate generation and statistics-based pruning methods. A hybrid solution HCA-GORDIAN combines the advantages of GORDIAN and our new algorithm HCA, and it outperforms all previous work in many situations.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database applications

General Terms

Algorithms

Keywords

unique, apriori, functional dependency, key discovery

1. UNIQUE COLUMN COMBINATIONS

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

DEFINITION 1. *Given a relational database schema* $R = \{C_1, C_2, \ldots, C_m\}$ *with columns* C_i *and an instance* $r \subseteq C_1 \times \ldots \times C_m$*, a column combination* $K \subseteq R$ *is a* unique*, iff*

$$\forall t_1, t_2 \in r: \ (t_1 \neq t_2) \Rightarrow (t_1[K] \neq t_2[K])$$

Unique discovery has high significance in several data management applications, such as data modeling, anomaly detection, query optimization, and indexing. Discovered uniques are good candidates for primary keys of a table. Therefore some literature refers to them as "candidate keys" [8]. The term "composite key" is also used to highlight the fact that they comprise multiple columns [9]. We want to stress that the detection of *uniques* is a problem that can be solved exactly, while the detection of *keys* can only be solved heuristically. Uniqueness is a necessary precondition for a key, but only a human expert can "promote" a unique to a key, because uniques can appear by coincidence for a certain state of the data. In contrast, keys are consciously specified and denote a schema constraint.

An important property of uniques and keys is their minimality. Minimal uniques are uniques of which no strict subsets hold the uniqueness property:

DEFINITION 2. *A unique* $K \subseteq R$ *is* minimal*, iff*

$$\forall K' \subset K: \ (\exists t_1, t_2 \in r: \ (t_1[K'] = t_2[K']) \wedge (t_1 \neq t_2))$$

In principle, to identify a column combination K of fixed size as a unique, all tuples t_i must be scanned. A scan has a runtime of $O(n)$ in the number n of rows. To detect duplicate values, one needs either a sort in $O(n \log n)$ or a hashing algorithm that needs $O(n)$ space. Non-uniques are defined as follows:

DEFINITION 3. *A column combination* K *that is not a unique is called a* non-unique*.*

Discovering all uniques of a table or relational instance can be reduced to the problem of discovering all minimal uniques. Every superset of a minimal unique is also a unique. Hence, in the rest of this paper the discovery of all uniques is synonymously used for discovering all minimal uniques. The exponential complexity is caused by the fact that for a relational schema $R = \{C_1, \ldots, C_m\}$, there are $2^m - 1$ subsets $K \subseteq R$ that can be uniques. Actually, even the result size of the problem can be exponential. In the worst case, there can be $\binom{m}{\frac{m}{2}}$ minimal uniques, each consisting of $\frac{m}{2}$ columns.

The contributions of this paper toward efficient unique detection are:

1. We analyze, discuss, and categorize existing algorithms and their strengths and weaknesses.
2. We introduce the new algorithm HCA and show how a bottom-up algorithm can be optimized by applying the concept of apriori candidate generation, statistics-driven pruning, and ad-hoc inference of FDs.
3. Furthermore, we present an elegant combination of HCA with the well-known GORDIAN algorithm [9], named HCA-GORDIAN, gaining even more efficiency.

2. RELATED WORK

Although the topic of finding or inferring composite keys and functional dependencies appeared ever since there are

*A full version of this paper is available at [1]

CIKM'11, October 24–28, 2011, Glasgow, Scotland, UK.

relational databases, there are only a few known approaches to the problem of discovering all minimal uniques of a table. These are discussed in detail in Sec. 3. In the broader area of meta data discovery however, there is much work related to the discovery of functional dependencies (FD). In fact, the discovery of FDs is very similar to the problem of discovering uniques, as uniques functionally determine all other individual columns within a table. There are several approaches for FD discovery [3, 6]; some include approximative solutions [5, 7]. Most new ideas in this research field follow either an Apriori or level-wise partitioning approach and require exponential runtime.

On the other hand, knowledge of FDs can be exploited for runtime-efficient unique discovery. Saeidian and Spencer present an FD-based approach that supports unique discovery [8]. They showed that given a minimal set of FDs , any column that appears only on the left side of the given FDs must be part of all keys and columns that appear only on the right side of the FDs cannot be part of any key. This insight cannot be used in the context of our work, as we assume no prior knowledge of functional dependencies, indexes, or semantic correlations. However, in Sec. 4 we show that our new algorithm HCA is able to infer some FDs on the fly and use them for apriori classification of some column combinations.

3. ALGORITHMIC FOUNDATIONS

In this section, the most important approaches to unique discovery are introduced, distinguishing two different classes: Row-based algorithms are based on a row-by-row scan of the database for all column combinations. The second class, column-based algorithms, contains algorithms that check the uniqueness of a column combination on all rows at once. Such column combinations are generated iteratively and each of them is checked only once.

Gordian: A Row-based Approach. Row-based processing of a table for discovering uniques requires multiple runs over all column combinations as more and more rows are considered. It benefits from the intuition that non-uniques can be detected without considering all rows of a table. A recursive unique discovery algorithm that works this way is Gordian [9]. The algorithm consists of three parts: *(i)* Preorganize table data in form of a prefix tree. *(ii)* Find maximal non-uniques by traversing the prefix tree. *(iii)* Compute minimal uniques from maximal non-uniques. The prefix tree has to be stored in main memory. Each level of the tree represents one column of the table whereas each branch stands for one distinct tuple. Non-unique discovery is performed by a depth-first traversal of the tree for discovering maximum repeated branches, which constitute maximal non-uniques. Maximal non-uniques can be defined as follows:

Definition 4. *A non-unique $K \subseteq R$ is* maximal*, iff all of its strict supersets $K' \supset K$ are unique.*

After the discovery of all maximal non-uniques, Gordian computes all minimal uniques by generating minimal combinations that are not covered by any of the maximal non-uniques. In [9] it is stated that this step needs only quadratic time in the number of minimal uniques, but the presented algorithm implies cubic runtime. The generation of minimal uniques from maximal non-uniques marks a serious bottleneck of the algorithm in case of large numbers of maximal non-uniques. Indeed, our experiments showed that in most cases the unique generation dominates the entire algorithm. Furthermore, the approach is limited by the available main memory and must be used on samples for approximate solutions when dealing with large data sets.

Apriori: A Column-based Approaches. The problem of finding minimal uniques is comparable to the problem of finding frequent itemsets [2]. The well-known Apriori approach is applicable for minimal unique discovery, working bottom-up as well as top-down. With regard to the powerset lattice of a relational schema the Apriori algorithms generate all relevant column combinations of a certain size and verify those at once. Figure 1 illustrates the powerset lattice for the running example in Tab. 1. The effectiveness and theoretical background of those algorithms is discussed by Giannela and Wyss [4]. They call their family of algorithms "Apriori-based", while in fact they make only minimal use of the Apriori idea.

first	last	age	phone
Max	Payne	32	1234
Eve	Smith	24	5432
Eve	Payne	24	3333
Max	Payne	24	3333

Table 1: Example data set

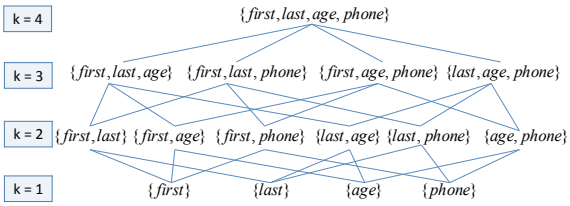

Figure 1: Powerset lattice for the example table

Bottom-up unique discovery indicates here that the powerset lattice of the schema R is traversed beginning with all 1-*combinations* to the top of the lattice, which is the $|R|$-*combination*. The prefixed number k of k-*combination* indicates the size of the combination. The same notation is used for the more specific terms k-*unique* and k-*non-unique*. The algorithm begins with checking the uniqueness of all individual columns. If a column is a unique, it will be added to the set of uniques, and if not it will be added to the list of *1-non-uniques*. The next iteration steps are based on the so-called candidate generation. A k-*candidate* is a potential k-*unique*. In principle, all possible k-*candidates* need to be checked for uniqueness. Effective candidate generation leads to the reduction of the number of uniqueness verifications by excluding apriori known uniques and non-uniques.

4. THE HCA APPROACH

In the following, we introduce the Histogram-Count-based Apriori Algorithm (HCA), an optimized bottom-up algorithm, which outperforms Gordian given a threshold of minimum average distinctness. The algorithm is based on the bottom-up algorithm presented in 3. We improve the algorithm by applying an efficient candidate generation, consideration of column statistics as well as ad-hoc inference and use of FDs. Finally, we describe how the advantages of our approach can be combined with advantages provided by Gordian for a hybrid solution.

4.1 Efficient Candidate Generation

Candidate generation is a crucial point for both bottom-up and top-down approaches. The more candidates can be

pruned apriori, the fewer uniqueness checks have to be performed and the better runtime will be achieved. Given a set of *k-non-uniques*, the naïve approach for generating $(k+1)$-*candidates* is to add non-contained 1-*non-uniques* to a *k-non-unique* [4]. The disadvantage of this approach is that repeated candidates may be generated: Given the example in Tab. 1, the combination {*first,phone*} would be identified as unique after the second pass of the algorithm. In the same pass, the combinations {*first,last*}, { *first,age*}, {*last,age*}, {*last,phone*}, and {*age,phone*} would be identified as non-uniques. In the third pass, the naive candidate generation would generate the 3-*candidates* including {*first,last,age*} by adding *age* to {*first,last*} , {*first,age,last*} by adding *last* to {*first,age*}, and {*last,age,first*} by adding *first* to {*last,age*}. These candidates are equal sets and their generation leads to unnecessary runtime overhead.

Furthermore, candidate generation faces another significant problem. Considering the running example, the generated 3-*candidates* would include {*first,last,phone*} by adding *phone* to { *first,last*} and {*first,age,phone*} by adding *phone* to {*first,age*}. Knowing that {*first,phone*} is a minimal unique, {*first,last,phone*} and {*first,age,phone*} are redundant uniques and their verification is futile.

Our candidate generation (Alg. 1) benefits from all optimizations of the classical apriori candidate generation in the context of mining association rules [2], so that repeated and redundant candidates are indeed pruned apriori.

The intuition behind our candidate generation is that a $(k+1)$-*unique* can only be a minimal iff all of its strict subsets are non-uniques. In other words, a $(k+1)$-*unique* can only be a minimal if the set of *k-non-uniques* contains all of the candidate's *k-subsets*. Because a minimality check is much cheaper than a verification step, we perform this step already within candidate generation. This is done by creating the union of every two *k-non-uniques* that are equal in the first $k-1$ columns (line 5). For the correctness of this operation, it is necessary that the columns are sorted lexicographically. We illustrate sorted sets using square brackets: $[C_1, C_2 \ldots]$. The sorting can be achieved by considering the order during the generation. 2-*candidates* are generated by cross-combination of all 1-*non-uniques*. Each new combination is sorted by a less-equal comparison of its two members. In candidate generations of later passes, the sorting is maintained by retaining the first $k-1$ columns in the preexisting order and a single comparison of the non-equal kth column of the two combined *k-non-uniques* (line 8).

A $(k+1)$-*combination* is not generated if there are no two *k-non-uniques* that conform exactly in the first $k-1$ columns. This is correct because it indicates that one *k-subset* is not a non-unique. Regarding our example from before, the redundant candidate [*first,last,phone*] would not be generated because it requires the occurrence of the sorted subsets [*first,last*] and [*first,phone*] as *k-non-uniques* that equal in the first column. However, [*first,phone*] as a previously discovered unique is missing.

A final minimality check on all remaining candidates prunes all redundant $(k+1)$*uniques*. Due to the inherent sortation of the non-uniques, the second important benefit of our candidate generation is the avoidance of repeated candidates.

4.2 Statistics-based Pruning

Real-world data contains semantic relations between column entries, such as correlations and functional dependencies (FDs). Knowledge of such relations and dependencies can be used to reduce the number of uniqueness checks. Unfortunately, these dependencies are usually not known. Based on retrieved count-distinct values and value frequencies, HCA is able to discover some FDs on the fly. In addition, knowledge of the number of distinct values of column combination and their value distribution allows further pruning by apriori non-unique detection.

Algorithm 1 CANDIDATEGEN

Require: *nonUniques* of size k
Ensure: *candidates* of size $k+1$
1: **for** $i \leftarrow 0$ to $|nonUniques| - 1$ **do**
2: **for** $j \leftarrow i+1$ to $|nonUniques| - 1$ **do**
3: *non-unique1* $\leftarrow$ *nonUniques*$[i]$
4: *non-unique2* $\leftarrow$ *nonUniques*$[j]$
5: **if** *non-unique1*$[0 \ldots k-2]$ = *non-unique2*$[0 \ldots k-2]$ **then**
6: *candidate* $\leftarrow$ new $k+1$-sized list
7: *candidate* $\leftarrow$ *non-unique1*$[0 \ldots k-2]$
8: **if** *non-unique1*$[k-1]$ <*non-unique2*$[k-1]$ **then**
9: *candidate*$[k-1] \leftarrow$ *non-unique1*$[k-1]$
10: *candidate*$[k] \leftarrow$ *non-unique2*$[k-1]$
11: **else**
12: *candidate*$[k-1] \leftarrow$ *non-unique2*$[k-1]$
13: *candidate*$[k] \leftarrow$ *non-unique1*$[k-1]$
14: **end if**
15: **if** ISNOTMINIMAL(*candidate*) **then**
16: **continue**
17: **end if**
18: *candidates*.ADD(*candidate*)
19: **end if**
20: **end for**
21: **end for**
22: **return** *candidates*

HCA is based on a hybrid verification scan that retrieves either the number of distinct values and the histogram of value frequencies of a combination or only the number of distinct values. A candidate is a unique if it contains as many distinct values as there are tuples in the table or if all value frequencies are 1. Regarding our example, column *last* contains the frequencies 1 and 3 for "Smith" and "Payne" respectively. It is a non-unique, because one value frequency is above 1. The retrieval of the histogram is still in $O(n \cdot log(n))$, because retrieving distinct count values and value frequencies need only a sort and a followup scan as it is needed by the duplicate detection approach.

Ad-Hoc Inference of Functional Dependencies. The first benefit of the count-based approach is that FDs can be identified. A functional dependency $X \rightarrow A$ allows us to conclude uniqueness statements: Given combinations $X, Y \subseteq R$ and a column $A \in R$, $\{X, Y\}$ is a unique if $\{A, Y\}$ is a unique and $X \rightarrow A$. In addition, if $\{A, X\}$ is a non-unique and $A \rightarrow B$, then $\{B, X\}$ is also a non-unique. These statements hold because the dependent side of an FD contains at most as many distinct values as the determinant side. For a column combination X and a column A, it holds $X \rightarrow A$ iff the number of distinct values of X equals the number of distinct values in the combination $\{A, X\}$.

HCA, illustrated in Alg. 2, retrieves those dependencies for all 1-*non-uniques* that are contained by the verified 2-*non-uniques*(line 35). In later iterations, for each member of a *k-candidate* it is scanned whether the column is part

of a discovered FD and if so which of the previously defined conclusions can be applied for the combination with the substituted member. So, it is possible to skip scans of *k-candidates* that were apriori classified through FDs.

The FD-based pruning takes place after each verification of a current candidate by looking for all substitutions that are possible using an existing FD (lines 23 and 28). The futility check in line 14 is performed to omit candidates that were already covered by FDs. Regarding our running example, it holds *phone* → *age*. Thus, knowing that {*first,phone*} is a non-unique, {*age,first*} must be a non-unique, too. On the other hand knowing that {*age, first,last*} is a unique {*first,last,phone*} must be a unique, too.

Algorithm 2 HCA Algorithm

```
Require: m columns
Ensure: Uniques
 1: for currentColumn in columns do
 2:   if isUnique(currentColumn) then
 3:     Uniques.add(currentColumn)
 4:   else
 5:     nonUniqueColumns.add(currentColumn)
 6:     storeHistogramOf(currentColumn)
 7:   end if
 8: end for
 9: currentNonUniques ← nonUniqueColumns
10: for k ← 2 to |nonUniqueColumns| do
11:   k-candidates ←candidateGen(currentNonUniques)
12:   currentNonUniques ← new empty list
13:   for candidate in k-candidates do
14:     if isFutile(candidate) then
15:       continue;
16:     end if
17:     if prunedByHistogram(candidate) then
18:       currentNonUniques.add(candidate)
19:       continue;
20:     end if
21:     if isUnique(candidate) then
22:       Uniques.add(candidate)
23:       for each FD k-candidate→ candidate do
24:         Uniques.add(k-candidate)
25:       end for
26:     else
27:       currentNonUniques.add(candidate)
28:       for each FD candidate → k-candidate do
29:         currentNonUniques.add(k-candidate)
30:       end for
31:       storeHistogramOf(candidate)
32:     end if
33:   end for
34:   if k = 2 then
35:     retrieveFDs()
36:   end if
37: end for
38: return Uniques
```

Count- and Histogram-based Pruning. Another benefit of the count- and histogram-based approach is the apriori identification of non-uniqueness of a *k-candidate* by considering the value frequencies of its combined $(k-1)$-*non-unique* subsets. The union of two non-unique combinations cannot be a unique if the product of the count-distinct values of these combinations is below the instance cardinality. In a more general case, it is sufficient to identify a value within one of the $(k-1)$-*non-uniques* that has a higher frequency than the number of distinct values within the other $(k-1)$-*non-unique*. In our running example, the 2-*candidate* {*last,age*} can be pruned because the value frequency of "Payne" is 3 and therefore higher than the number of distinct values in *age*, which is only 2. In case the value with the highest frequency equals the number of distinct values of the other $(k-1)$-*non-unique*, we compare the next highest value frequency with the count distinct value of a modified view of the other $(k-1)$-*non-unique*. In the modified view each frequency is decreased by 1 so that it can be assumed that each distinct value was combined once with the more frequent value.

This approach has two drawbacks: *(i)* Such constellation of value frequencies appears only in early passes of the algorithm; *(ii)* histograms must be stored in memory. The remedy for the two drawbacks is to perform histogram retrieval only for single columns (line 6) and to store only count-distinct values in later passes. In line 17, for each generated *k-candidate*, it is checked whether one of the two combined $(k-1)$-*non-uniques* has a lower count-distinct value than a value frequency of the additional kth column. Note, for an apriori identified non-unique there will be no count-distinct value that can be used in the next pass. Thus, the pruning takes place in at most every second pass of the algorithm.

4.3 Combination of GORDIAN and HCA

In Sec. 3, we stated that the unique-generation part of the Gordian algorithm is inefficient if the number of discovered non-uniques is high. By profiling the runtime of Gordian we could identify the unique generation as the bottleneck. At the same time the non-unique discovery consumed only a fraction of the runtime. Thus, an intriguing idea is to interlace the non-unique discovery of Gordian with the candidate generation of HCA.

We combined Gordian with HCA by performing the non-unique discovery of Gordian on a smaller sample of the table and executing HCA on the entire table. Non-uniques discovered within a sample of a relational instance are also non-uniques for the complete instance and can be used for pruning candidates during the HCA part of the algorithm. It is thus possible to smooth the worst case of the bottom-up algorithm by skipping non-uniques identified by Gordian, and simultaneously to avoid Gordian's bottleneck of unique generation.

5. EVALUATION

We tested HCA and HCA-Gordian against Gordian itself and the basic bottom-up, top-down and Hybrid Apriori approaches introduced in Sec. 3. We implemented two different versions of the bottom-up algorithm: The approach identified by "BU Apriori" uses the candidate generation in Alg. 1, and a naïve approach labeled as "Naïve BU" generates candidates without pruning redundant non-minimal uniques. The "Naïve BU" is actually the implementation of the bottom-up algorithm proposed by Giannella and Wyss [4].

The algorithms were tested on synthetic data as well as real-world data. All algorithms are self-implemented in Java 6.0 on top of a commercial relational database. The experiment platform had the following properties:

- Windows Vista (32 Bit) Business™
- Pentium (R) Dual-Core CPU E5200 @2.50 GHz
- JRE limited to 1 GB RAM

5.1 Synthetic Data

We compared the algorithms with regard to increasing number of rows, columns, average distinctness. Additional important parameters for the algorithms are the number of uniques and their average size. Unfortunately, both values are only available after a successful completion of one algorithm. The generation of random data with specific number and size of uniques is probably as hard as the problem of discovering the minimal uniques and is an important challenge for future work. Nevertheless we also analyze these values when looking at the runtime of each algorithm.

Influence of Number of Rows. We generated multiple tables with 20 columns differing in the row-count. Note, the bigger the table, the lower the average distinctness might be, because the possibility of repeated values increases with increasing number of tuples. So, for the table with 10,000 tuples the average distinctness is 47%, while for the table with 200,000 tuples the value is 3%. Figure 2 illustrates the runtime of all algorithms with regard to row-counts between 10,000 and 200,000. In addition to the number of tuples, the number of uniques for each data set is denoted below the number of rows.

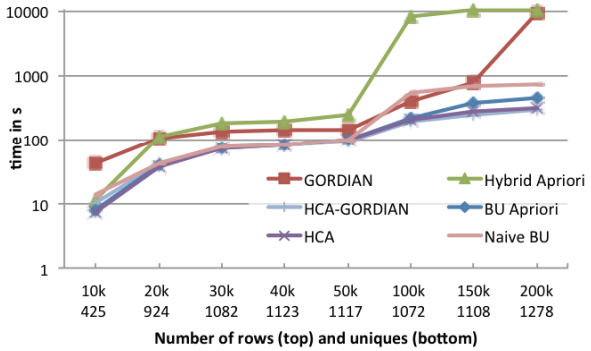

Figure 2: Runtime with respect to increasing number of rows on datasets with 20 columns

The Top-Down Algorithm is omitted in the diagram because its runtime was by magnitudes worse than Gordian. As all uniques in these experiments are combinations of only few columns, a bottom-up approach discovers all of them, earlier. The Hybrid Apriori Algorithm performs worse than the bottom-up approaches, but still better than the Top-Down Algorithm. The HCA-Gordian performed clearly better than Gordian. This is probably due to the fact that the relatively high number of uniques slows down the unique generation step of Gordian, which is avoided in HCA-Gordian. HCA-Gordian performed the preprocessing with Gordian for non-unique discovery always on a 10,000 tuple sample. On the data sets with more than 100,000 tuples, both HCA and HCA-Gordian perform at least 10% better than BU Apriori, which does not perform HC-based pruning. Note, the bottom-up approaches with our efficient candidate generation performed always at least 25% better than the Naïve BU. On the datasets with more than 100,000 rows, the performance gain was above 60%. Furthermore, all bottom-up algorithms outperform Gordian on all of these data sets.

Table 2 denotes the maximum memory usage of the algorithms on the table with 100,000 rows. Gordian performs clearly worse than all other algorithms. The Apriori approaches perform best and are nearly equal. The memory usage of the HCA-Gordian algorithm is due to the prefix tree on sample data higher than HCA.

Algorithm	Memory Usage
Gordian	534 MB
HCA-Gordian	71 MB
HCA	20 MB
Bottom-Up Apriori	19 MB
Hybrid Apriori	20 MB

Table 2: Memory usage for 100,000 tuples, 20 columns and 7.5% average distinctness

Influence of Number of Columns. Theoretically, the runtime of any algorithm is exponential in the number of columns, in the worst case. The algorithms have been tested on data sets consisting of 15 to 25 columns. The data sets each consist of 10,000 tuples and hold an average distinctness of about 5%. The experimental results are presented in Fig. 3. As expected, the runtime of all algorithms increases with the number of columns, but the incline of the curves is far smaller than exponential. Gordian again performs worse than all bottom-up approaches. The remarkable runtime decrease on the data set with 20 columns is due to the decrease of the number of uniques from 1,712 to 1,024. This is a good example for the unpredictability of the runtime of Gordian because of its high dependence on the number of existing uniques. For the datasets with more than 21 columns, the runtime of Gordian exploded.

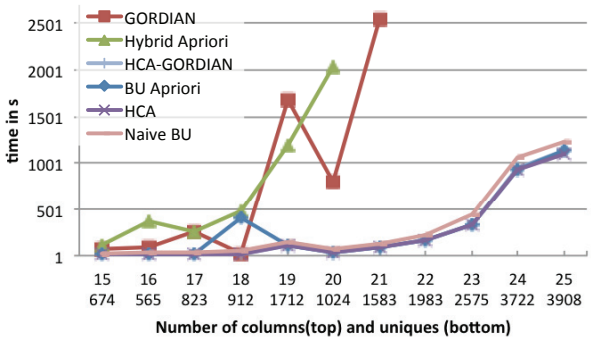

Figure 3: Runtime with respect to column numbers

Influence of Average Distinctness. The higher the average distinctness of all columns, the smaller is the size of minimal uniques – in the extreme case, already individual columns are unique. Thus, also the number of uniques is expected to be low. On the other hand, if the average distinctness is very low, minimal uniques become very large – in the extreme case only the entire relation is a unique. Again, the number of uniques is expected to be low. In between, the number of uniques is expected to be higher. This behavior can be observed in the generated data sets: Figure 4 shows the observed average numbers of minimal uniques for different average distinctnesses (five datasets each) and the observed average runtimes for data sets with 15 columns and 10,000 tuples.

Considering Fig. 4, all algorithms perform better on data with high average distinctness. For Gordian, the opposite case is expected as it is based on discovering non-uniques: Lower distinctness should result in faster discovery of non-uniques and better runtime for Gordian. However, low distinctness is accompanied by higher number of uniques and

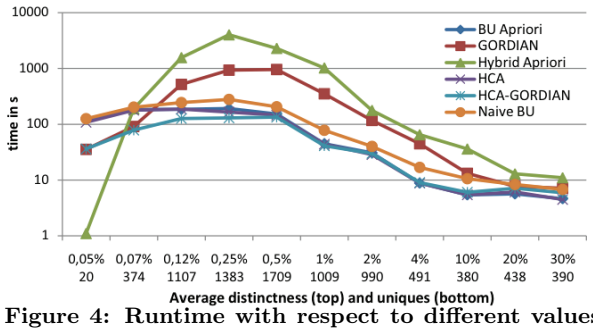

Figure 4: Runtime with respect to different values of average distinctness

non-uniques, which leads to more overhead during unique generation. This can be observed among the distinctness values between 0.12% and 2%, where the average unique size was 7, which is about one half of the number of columns. In fact, the Bottom-Up Algorithm, HCA, HCA-Gordian, and even the Naïve BU performed better in this range. Vice versa, the Hybrid Apriori Algorithm performed worst. Regarding the distinctness range below 0.12%, Gordian outperforms the bottom-up algorithms. The Hybrid Apriori Algorithm outperforms all other algorithms in this range due to the fact that the average size of uniques is 12 and Hybrid Apriori checks uniques of this size earlier than all other algorithms. That means that the Top-Down Apriori Algorithm would have performed even better. HCA-Gordian consistently performs better or at least as good as HCA and Gordian.

5.2 Real World Data

Real world data may differ in its nature from domain to domain. Table 3 lists four real world tables that were downloaded from the data collecting website `factual.com`. Table 4 presents the runtime results for these data sets. HCA and HCA-Gordian outperformed Gordian on all tables where only one unique was to be discovered, because their bottom-up approach discovers single column uniques very fast. Especially the experiment on the "National File" table shows the disadvantage of Gordian with regard to scalability, because the prefix tree did not fit into 1GB main memory. The NFL Stats data set that contains also multi-column uniques shows that there are data where Gordian still performs best. However, the hybrid solution HCA-Gordian is not remarkably worse.

Table	tuples	columns	uniques
US places	195,762	19	1
National file	1,394,725	20	1
NFL Stats	42,589	14	10

Table 3: Real world tables with statistics

Table	Gordian	HCA-Gordian	HCA
US places	64.13 s	15.389 s	9.232 s
National file	too large	114.919 s	130.239 s
NFL Stats	79.26 s	86.345 s	263.513 s

Table 4: Real world tables with runtime results

Summary. All algorithms show strengths and weaknesses for different value distributions, size and number of uniques. Efficient candidate generation leads to remarkable runtime improvement of the bottom-up algorithms. The HC-based pruning methods improve the algorithms on large data sets with low average distinctness. HCA-Gordian is a significant improvement of the basic HCA having large tables. HCA-Gordian performs better than Gordian when the number of detected non-uniques is high. Gordian performs best on data with low average distinctness and small number of uniques. The HCA approaches are much more memory efficient than Gordian.

6. CONCLUSIONS AND FUTURE WORK

In this paper we elaborated the concepts of uniques and non-uniques, the effects of their size and numbers, and showed strengths and weaknesses of existing approaches. We introduced the new bottom-up algorithm HCA, which benefits from apriori candidate generation and data- and statistic-oriented pruning possibilities. Furthermore, we showed a simple way of combining HCA and Gordian for even better runtime results. A more detailed analysis of these approaches and their optimizations is provided as a technical report [1]. The results of this paper constitute further open directions. The most important issue for further research is *approximate unique discovery*. As HCA is a statistics-based approach, it allows further optimizations based on statistics-driven heuristics for approximate solutions.

Another open issue is the need for a flexible and efficient data generator that should be able to generate a table that contains a fixed number of uniques of a certain size and holds specific value distributions for all columns. Thus, it is possible to evaluate and benchmark algorithms for special cases that might occur. Finally, the recent proposals for column stores call for unique discovery solutions that benefit from features of a column-based DBMS. Here, the column-based approach HCA is a promising candidate.

7. REFERENCES

[1] Z. Abedjan and F. Naumann. Advancing the discovery of unique column combinations. Technical Report 51, Hasso-Plattner-Institute Potsdam, Germany, 2011.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 487–499, San Francisco, CA, 1994.

[3] P. A. Flach and I. Savnik. Database dependency discovery: a machine learning approach. *AI Commun.*, 12:139–160, August 1999.

[4] C. Giannella and C. Wyss. Finding minimal keys in a relation instance. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.7086`, 1999. Last accessed on 2010-09-29.

[5] Y. Huhtala, J. Kaerkkaeinen, P. Porkka, and H. Toivonen. TANE: an efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100–111, 1999.

[6] M. Kantola, H. Mannila, K.-J. Räihä, and H. Siirtola. Discovering functional and inclusion dependencies in relational databases. *International Journal of Intelligent Systems*, 12:591Ű–607, 1992.

[7] V. Matos and B. Grasser. SQL-based discovery of exact and approximate functional dependencies. *SIGCSE Bull.*, 36(4):58–63, 2004.

[8] H. Saiedian and T. Spencer. An efficient algorithm to compute the candidate keys of a relational database schema. *The Computer Journal*, 39(2):124–132, 1996.

[9] Y. Sismanis, P. Brown, P. J. Haas, and B. Reinwald. Gordian: efficient and scalable discovery of composite keys. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 691–702, 2006.