Advancing the Discovery of Unique Column Combinations

Ziawasch Abedjan, Felix Naumann

Technische Berichte Nr. 51

des Hasso-Plattner-Instituts für
Softwaresystemtechnik
an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam | 51

Ziawasch Abedjan | Felix Naumann

Advancing the Discovery of Unique Column Combinations

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.de/ abrufbar.

Universitätsverlag Potsdam 2011
http://info.ub.uni-potsdam.de/verlag.htm

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URL http://pub.ub.uni-potsdam.de/volltexte/2011/5356/
URN urn:nbn:de:kobv:517-opus-53564
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-53564

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-148-6

Advancing the Discovery of Unique Column Combinations

Ziawasch Abedjan Felix Naumann
firstname.lastname@hpi.uni-potsdam.de

Hasso Plattner Institute, Potsdam, Germany

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 [19] and "Apriori-based" algorithms [5] 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 significantly outperforms all previous work in many situations.

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*

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

Discovered uniques are good candidates for primary keys of a table. Therefore some literature refers to them as as "candidate keys" [17]. The term "composite key" is also used to highlight the fact that they comprise multiple columns [19]. 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. Both approaches need only subquadratic runtime. Of course, a scan can be aborted with a "non-unique" result as soon as a duplicate value is detected. 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 and can be readily identified once the minimal uniques are known. Hence, in the rest of this paper the discovery of all uniques is synonymously used for discovering all minimal uniques.

The problem of discovering one minimal unique with a maximal number of columns is NP-complete in the number of columns [8]. Also, the problem of finding the quantity of all minimal uniques is #P-hard. Both complexity classes indicate exponential runtimes for the discovery of all minimal uniques in a table. 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.

1.1 Usage of uniques

Uniques help to understand the structure and the semantic properties of tabular data. Unique discovery is part of the metadata discovery methods for data profiling and has high significance in several data management applications, such as data modeling, anomaly detection, query optimization, and indexing. These applications are relevant for data architecture, data design, and database management. However, tables may contain unknown uniques for several reasons: experimental origins of the data, lack of support for checking uniqueness constraints in the host system for fear that checking such constraints would impede database performance, or a simple lack of application domain knowledge within the development team. These reasons are, for instance, quite frequently met in Life Science databases. Among the very popular genome database Ensembl [4] and the Protein Data Bank [2], unique and key constraints are only sporadically available and none of them refer to multiple columns. Understanding and integrating such databases greatly benefit from methods to discover uniques.

Furthermore, identification of uniques plays an important role in data mining: first, many data mining approaches could be more efficient by concentrating on uniques instead of all columns of a table. In the context of bioinformatics, the

discovery of uniques may lead to detection of unknown principles between protein and illness origins [12, 18]. The second point of coherence is related to the process of unique discovery itself. The discovery of all minimal uniques within a relational instance using only polynomial time in the number of columns *and number of uniques* is stated as an open problem in data mining by Mannila [13]. In Sec. 2, where foundations are presented, the classical data mining algorithm Apriori [1] is introduced as a reasonable way of approaching the minimal unique problem. We complete this approach by adapting the apriori candidate generation for unique discovery solution.

1.2 Related Work

Although the topic of finding or inferring composite keys and functional dependencies appeared ever since there are 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. 2. 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. Furthermore, minimality plays an important role in both concepts. There are several approaches for FD discovery [3, 11]; some include approximative solutions [9, 10, 16]. Most new ideas in this research field follow either an Apriori or level-wise partitioning approach and require exponential runtime. Mannila proved that the inference of FDs has exponential runtime in the number columns as lower bound [14].

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 by identifying columns that are definitely part of all uniques and columns that are never part of any unique [17]. They showed that given a minimal set of FDs (only one attribute appears on the right side of each FD; the left side of each FD is irreducible, i.e., removing any part results in a non-equivalent set; entirely removing an FD also results in a non-equivalent set), 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. 3 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.

Finally, there is the research field of discovering approximate keys within semi-structured data. Grahne presents an Apriori approach for discovering approximate keys within XML data [6]. Their algorithm evaluates discovered key candidates by the metrics support and confidence. The approaches that are introduced as foundations in Sec. 2.2 follow a similar intuition.

1.3 Contributions

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 bottom-up algorithm HCA, which outperforms existing algorithms by multiple factors given a threshold of minimum average distinctness of the columns. We show how a bottom-up brute force algorithm can be highly optimized by applying the concept of apriori candidate generation, data- and statistics-driven pruning, and ad-hoc inference of FDs.
3. Furthermore, we present an elegant combination of HCA with the well-known Gordian algorithm [19], named HCA-Gordian, gaining even more efficiency.
4. We show the advantages of our approaches HCA and HCA-Gordian in a detailed evaluation comparing them to state-of-the-art algorithms on synthetic and real-world data.

The rest of this paper is organized as follows: In Sec. 2 the most relevant existing algorithms are described as foundations of our new algorithms. In Sec. 3 we introduce the new bottom-up algorithm HCA and the hybrid solution HCA-Gordian. We present a detailed evaluation of our algorithms in Sec. 4 and conclude in Sec. 5.

2 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.

2.1 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 [19]. The algorithm consists of three parts:

1. Preorganize table data in form of a prefix tree.
2. Find maximal non-uniques by traversing the prefix tree.
3. Compute minimal uniques from maximal non-uniques.

The prefix tree has to be created and stored in main memory. Each level of the tree represents one column of the table whereas each branch stands for

one distinct tuple. The traversal of the tree is based on the cube operator [7], which computes aggregate functions on projected columns. Non-unique discovery is performed by a depth-first traversal of the tree for discovering maximum repeated branches, which constitute maximal non-uniques. Maximality of 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 [19] it is stated that this step needs only quadratic time in the number of minimal uniques, but the presented algorithm implies cubic runtime, because for each considered non-unique the changed set of uniques must be *simplified* by removing redundant uniques. This simplification requires quadratic runtime in the number of entries in the actual set of uniques. As the number of minimal uniques is bounded linearly by the number s of maximal non-uniques [19], the runtime of the unique generation step is in $O(s^3)$.

GORDIAN benefits from the intuition that non-uniques can be discovered faster than uniques. Remember, non-unique discovery can be aborted as soon as one repeated value is discovered among the projections. The prefix structure of the data facilitates this analysis. It is stated that the algorithm is polynomial in the number of tuples for data with a Zipfian distribution of values. Nevertheless, in the worst case GORDIAN will have exponential 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. Although data may be compressed because of the prefix structure of the tree, the amount of processed data may still be too large to be maintained in main memory.

2.2 Column-based Approaches

The problem of finding minimal uniques is comparable to the problem of finding frequent itemsets [1]. 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 [5]. They call their family of algorithms "Apriori-based", while in fact they make only minimal use of the Apriori idea. In the following we describe briefly the features of their algorithms, which will serve as the foundation of our improved algorithms.

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

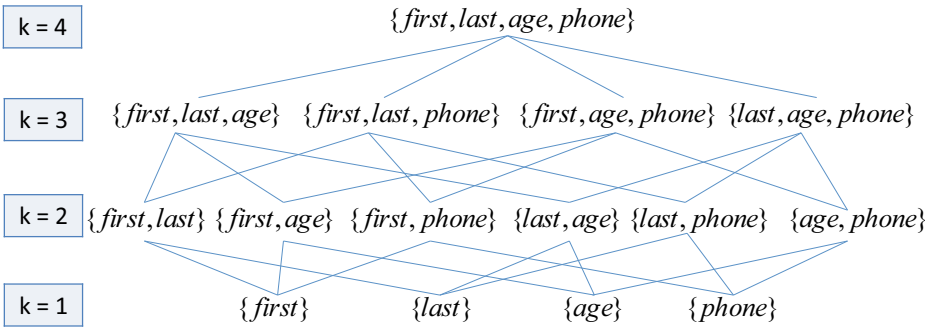

Fig. 1. Powerset lattice for the example table

Bottom-Up Apriori Bottom-up unique discovery is very similar to the Apriori algorithm for mining association rules. Bottom-up 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. E.g., if a column C is a unique and a column D is a non-unique, it is apriori known that the combination $\{C, D\}$ must be a unique. Thus, k-*candidates* are generated within the bottom-up approach by using the previously discovered $(k-1)$-*non-uniques*. In their solution, k-*candidates* with $k > 2$ are generated by extending the $(k-1)$-*non-uniques* by another non-unique column. After the candidate generation, each candidate is checked for uniqueness. If it is identified as a non-unique, the k-*candidate* is added to the list of k-*non-uniques*. If the candidate is verified as unique, its minimality has to be checked. The algorithm terminates when $k = |1\text{-}non\text{-}uniques|$. A disadvantage of this candidate generation is that redundant uniques and duplicate candidates are generated. In Sec. 3 we introduce a candidate generation

that remedies these two problem by adapting the apriori paradigm "*every subset of a frequent itemset is also a frequent itemset*" in data mining to unique discovery.

Top-Down and Hybrid Apriori The search for uniques within the powerset lattice can also work top-down, i.e., beginning with the largest *k-candidate*, which is the schema R. If the *k-candidate* is a non-unique, all of its subsets are also non-uniques. If the *k-candidate* is a unique it is still not guaranteed, whether it is minimal. Therefore, all of its $(k-1)$-sized subsets have to be checked for uniqueness. The same procedure would apply in turn to all those subsets. The algorithm terminates when the iteration for $k = 1$ finishes.

Both the Bottom-Up and the Top-Down Algorithm have exponential runtime in the worst case. However, the worst case of each algorithm is at the same time the best case for the other algorithm. The hybrid approach combines both algorithms [5]. The algorithm has a loop from $m-1$ down to $\lceil \frac{m}{2} \rceil$ where m is the number of 1-*non-uniques*. Column combinations of size k and $m-k$ are checked for each $k \in [\lceil \frac{m}{2} \rceil, m]$. By an additional communication channel between the top-down and the bottom-up part of the algorithm, additional pruning aspects are possible.

On every pass with $k > \frac{m}{2}$, the bottom-up part of the algorithm prunes those $(n-k)$-*candidates* that have no k-*unique* as a superset, which was previously discovered by the top-down part of the algorithm. Accordingly, the top-down part of the algorithm prunes k-*candidates*, when all of its $(n-k)$-sized subsets are supersets of $(n-k-1)$-*uniques*, detected by the bottom-up part in the previous pass. The hybrid algorithm remains exponential in the worst case.

None of the introduced algorithms so far solve Mannila's problem of discovering uniques within polynomial time. However, the Apriori approaches provide enough room for data driven optimizations and can be improved by a more effective candidate generation. In the following section, we introduce an improved algorithm combining the advantages of the Apriori and Gordian algorithms.

3 The HCA Approach

Gordian showed remarkable runtime and space efficiency compared with a brute-force algorithm in their experiments on real world and synthetic datasets [19]. In the following, we introduce the Histogram-Count-based Apriori Algorithm (HCA), an optimized bottom-up Apriori algorithm, which outperforms Gordian by multiple factors given a threshold of minimum average distinctness. In principle, the algorithm is based on the bottom-up algorithm presented in 2.2. We will show that the algorithm can be improved by applying additional pruning possibilities that are enabled 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.

3.1 Efficient Candidate Generation

Candidate generation for level-wise apriori approaches 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.

Bottom-up Candidate Generation 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*. This is the way candidates are generated by the bottom-up algorithm described by Giannella and Wyss [5]. 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 contain the same set of columns and their generation leads to unnecessary runtime overhead. Detection of repeated combinations requires the comparison of all generated combinations to each other by considering all contained columns of the compared combinations.

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 that are associated with the classical apriori candidate generation in the context of mining association rules [1], so that repeated and redundant candidates are indeed pruned apriori. A very similar approach is also used for mining inclusion dependencies [15].

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*. This is the adapted intuition of the classical Apriori algorithm. 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 6). For the correctness of this operation, it is necessary that the columns are sorted lexicographically. From here on 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 simple less-equal comparison of its two members. In candidate generations of later passes, the sorting

is maintained by retaining the first $k-1$ elements in the preexisting order and a single comparison of the non-equal kth element of the two combined *k-non-uniques* (line 9).

A $(k+1)$-*combination* is not generated if there are no two *k-non-uniques* that conform exactly in the first $k-1$ elements. 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 element. However, [*first,phone*] as a previously discovered unique is missing.

Unfortunately, the candidate [*age, first, phone*] is still generated, because [*age, first*] and [*age, phone*] are both known *k-non-uniques* and can be combined to a 3-*candidate*. Therefore, we perform a final minimality check on all remaining candidates that prunes all remaining redundant uniques. The minimality check can be performed in linear time in the number of already discovered minimal uniques, when using an additional bitmap representation of the column combinations. A simple *OR*-operation shows if a combination covers another or not. The bitmap representation of the uniques has already been proposed for Gordian. Due to the inherent sortation of the non-uniques, the second important benefit of our candidate generation towards the naive approach [5] is the avoidance of repeated candidates.

Top-down Candidate Generation While the paradigm for generating *k-candidates* in the bottom-up approach is that all $(k-1)$-sized subsets have to be non-uniques, the paradigm for the top-down approach is that all of the $(k+1)$-sized supersets of a *k-combination* must be uniques. Recall that the optimized bottom-up candidate generation generates *k-candidates*, knowing that at least two $(k-1)$-sized subsets are non-uniques. The optimized subsetting approach for the top-down approach works similar, using two $(k+1)$-sized supersets. Unfortunately, here generation of repeated combinations cannot be avoided, because for a *k-candidate* there may be up to $m-k$ $(k+1)$-sized supersets. Therefore, a subsequent check for repeated combinations is required.

3.2 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.

The most efficient way of scanning a combination for uniqueness is to look for duplicate projections in $O(n \cdot log(n))$. HCA is based on a hybrid verification scan

Algorithm 1 CANDIDATEGEN

Require: *nonUniques* of size k
Ensure: *candidates* of size $k+1$

```
1: candidates ← new empty list
2: for i ← 0 to |nonUniques| − 1 do
3:    for j ← i + 1 to |nonUniques| − 1 do
4:       non-unique1 ← nonUniques[i]
5:       non-unique2 ← nonUniques[j]
6:       if non-unique1[0 ... k − 2] = non-unique2[0 ... k − 2] then
7:          candidate ← new k + 1-sized list
8:          candidate ← non-unique1[0 ... k − 2]
9:          if non-unique1[k − 1] <non-unique2[k − 1] then
10:            candidate[k − 1] ← non-unique1[k − 1]
11:            candidate[k] ← non-unique2[k − 1]
12:         else
13:            candidate[k − 1] ← non-unique2[k − 1]
14:            candidate[k] ← non-unique1[k − 1]
15:         end if
16:         if isNotMinimal(candidate) then
17:            continue
18:         end if
19:         candidates.add(candidate)
20:      end if
21:   end for
22: end for
23: return candidates
```

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. However, the retrieval of these statistics always requires a complete scan in contrast to the duplicate detection approach, which aborts as soon as a duplicate is discovered. So, the trade-off is to execute scans that are more expensive for the retrieval of usable pruning information, but to gain runtime efficiency by significantly reducing the number of such scans by means of those apriori retrieved information.

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 single non-uniques that are contained by the verified 2-*non-uniques* in line 36. Note, the count-distinct values of identified 1- and 2-*non-uniques* are already known. Thus, we consider only single columns that are non-uniques. 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 later scans of *k-candidates*, which were apriori classified with the help of FDs. FDs with multiple columns on the left side are ignored because their identification requires the retention of all count distinct values of the already checked smaller combinations and too complex look-up structures.

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, as it is shown in the lines 24 and 29. The futility check in line 15 is performed to omit candidates that were already covered by FDs. Regarding our running example, it holds *phone* $\rightarrow$ *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.

Count- and Histogram-based Pruning Another benefit of the count- and histogram-based approach is the apriori identification of non-uniqueness of a *k*-

Algorithm 2 HCA Algorithm

Require: m columns
Ensure: *Uniques*

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

candidate by considering the value frequencies of its combined $(k-1)$-*non-unique* subsets. In fact, 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 7) and to store only count-distinct values in later passes. In line 18, 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. Nevertheless on relational instances with relatively low average distinctness, it leads to remarkable performance improvement towards the bottom-up approach without the histogram- and count-based prunings. This is illustrated among the experiment results in Sec. 4.

3.3 Combination of GORDIAN and HCA

In Sec. 2, we stated that the unique-generation part of the Gordian algorithm might be inefficient if the number of discovered non-uniques is high. Indeed, experiments presented in Sec. 4 show that unique discovery is always remarkably slow if many uniques are present. By profiling the runtime of Gordian we could identify the unique generation as the bottleneck. Note, the number of minimal uniques is linear in the number of maximal non-uniques. 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. Note that 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.

4 Evaluation

We tested HCA and its combination with Gordian (HCA-Gordian) against Gordian itself and the basic bottom-up, top-down and Hybrid Apriori approaches introduced in Sec. 2. 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 Apriori bottom-up algorithm proposed by Giannella and Wyss [5].

The algorithms were tested on synthetic data as well as real-world data. We used a self-implemented data generator that provides series of synthetic tables, each with focus on one important parameter, such as number of columns, number of rows, or average distinctness of columns. All algorithms, including Gordian, 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

4.1 Synthetic Data

We compared the algorithms with regard to increasing number of rows, increasing number of columns, and increasing average distinctness. Note, the generated tables contain no duplicate tuples. 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.

The Top-Down Algorithm is omitted in the diagram, because its runtime was by magnitudes worse than Gordian. The poor performance of the Top-Down Algorithm confirms the fact that the bottom-up algorithms perform best. 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

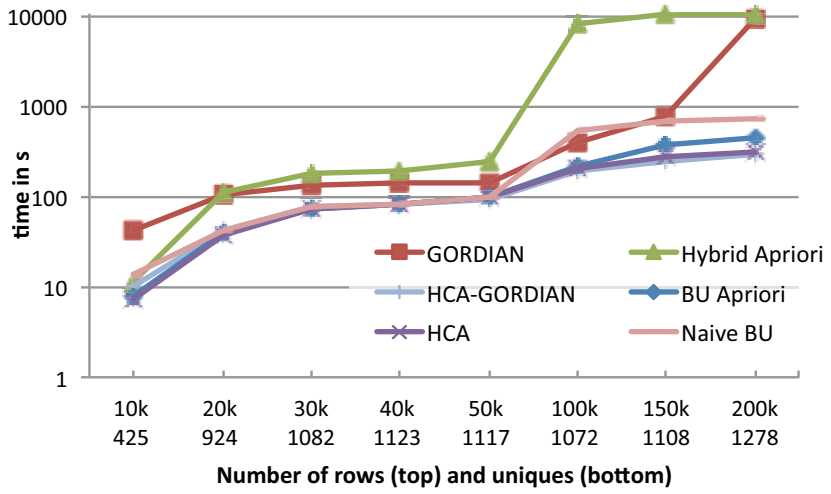

Fig. 2. Runtime with respect to increasing number of rows on datasets with 20 columns 7.5% average distinctness

worse than the bottom-up approaches, but still better than the omitted Top-Down Apriori 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.

Because of the logarithmic scale in Fig. 2, it is possible to further analyze the fastest algorithms: HCA-Gordian performed the preprocessing with Gordian for non-unique discovery always on a 10,000 tuple sample. This overhead did not lead to worsening of the algorithm's runtime in comparison to the standard HCA. In fact, there is no obvious runtime difference between HCA and HCA-Gordian among these data sets. The combination HCA-Gordian leads only to minor performance improvements over HCA if the processed data has relatively low distinctness. The following experiments, which focus on distinctness, confirm this statement. 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, because of the small sized uniques and relatively high distinctness.

Table 2 denotes the maximum memory usage of the algorithms with regard to 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 higher than HCA. This is

caused by the preprocessing with Gordian that needs to create a prefix tree on sample data.

Algorithm	Memory Usage
Gordian	534 MB
HCA-Gordian	71 MB
HCA	20 MB
Bottom-Up Apriori	19 MB
Naïve BU	20 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 The second parameter that influences the runtime of unique discovery algorithms is the 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. The runtime of the column-based approaches and HCA-Gordian increase steadily in the number of columns. All algorithm with the efficient candidate generation perform quite similar on these data sets. Even the Naïve BU performs only slightly worse than the optimized approaches.

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. The data sets consist of 15 columns and 10,000 tuples and differ in their average distinctness.

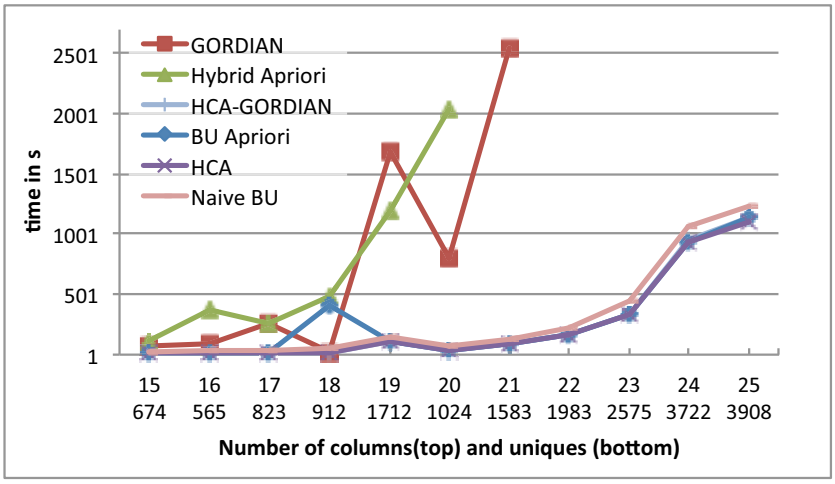

Fig. 3. Runtime with respect to column numbers

Considering Fig. 4, all algorithms perform better on data with high average distinctness. With respect to the Apriori algorithms, this can be explained by the fact that high distinctness results in smaller sized uniques and therefore fewer passes of the algorithm. 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 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. In this experiment scenario, there is no obvious performance gain of HC-based pruning towards simple Bottom-Up Apriori algorithms, because the number of rows is not high enough that pruning leads to significant runtime efficiency towards expensive table scans. HCA-Gordian consistently performs better or at least as good as HCA and Gordian.

4.2 Real World Data

Real world data may differ in its nature from domain to domain. It contains unpredictable value distributions and column correlations, such as FDs that are

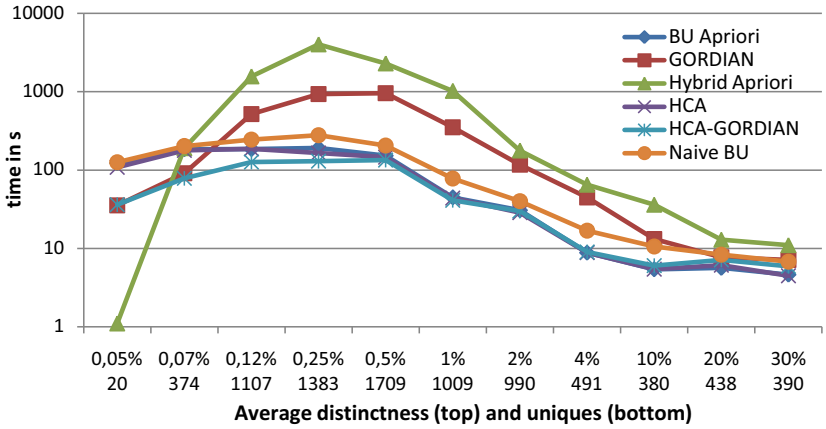

Fig. 4. Runtime with respect to different values of average distinctness on tables with 15 columns and 10,000 rows

not producible by a data generator. Table 3 lists four real world tables. The first three tables were downloaded from the data collecting website `factual.com`; the last was kindly provided by `film-dienst.de`. 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 and aborts the search if no other unique is available. 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 example with the NFL Stats that contains also multi-column uniques shows that there are data where Gordian still performs best – about four times faster than HCA. 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
Movies	55,988	33	1

Table 3. Real world tables with statistics

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 and the unique generation part of Gordian

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
Movies	24,050.79 s	658.029 s	2.223 s

Table 4. Real world tables with runtime results

dominates the algorithm. GORDIAN performs best on data with low average distinctness and small number of uniques. The HCA approaches are much more memory efficient than GORDIAN.

5 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. Based on the discovered potentials 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. We evaluated all algorithms on generated data as well as on real world data, demonstrating the advantages of our algorithms over existing approaches. As HCA is a statistics-based approach, it allows further optimizations based on statistics-driven heuristics for approximate solutions. HCA is also suited for sampling approaches and then faces the same restrictions as GORDIAN when it comes to the precision of the sampling-based results.

Despite the fact that Mannila's problem statement about creating an algorithm with polynomial runtime in the number of columns and minimal uniques is still an open problem, the results and insights of this paper constitute further open directions. The most important issue for further research is *approximate unique discovery*.

Another open issue is the need for a flexible and efficient data generator. At best, a data generator 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 most special cases that might occur and identify their strength and weaknesses more explicitly. Another important challenge in the field of evaluating unique discovery algorithms is to define more appropriate metrics for approximate algorithms. Especially with regard to minimal uniques, the metric should consider the size difference of wrongly identified minimal uniques and its superset or subset that is the actual minimal unique. So, the metric should be able to distinguish the retrieval of a non-unique, a non-minimal unique, and a minimal unique.

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. Indeed, the applicability and

performance of unique and FD discovery algorithms might be an interesting evaluation criterion when comparing the capabilities of column stores with relational databases.

References

1. 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.
2. H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The Protein Data Bank. *Nucleic Acids Res*, 28:235–242, 2000.
3. P. A. Flach and I. Savnik. Database dependency discovery: a machine learning approach. *AI Commun.*, 12:139–160, August 1999.
4. P. Flicek et al. Ensembl 2008. *Nucleic Acids Research*, 36(suppl 1):D707–D714, 2008.
5. 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.
6. G. Grahne and J. Zhu. Discovering approximate keys in XML data. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 453–460, New York, NY, 2002.
7. J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.
8. D. Gunopulos, R. Khardon, H. Mannila, and R. S. Sharma. Discovering all most specific sentences. *ACM Transactions on Database Systems (TODS)*, 28:140–174, 2003.
9. Y. Huhtala, J. Kaerkkaeinen, P. Porkka, and H. Toivonen. Efficient discovery of functional and approximate dependencies using partitions. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 392–401, Washington, DC, 1998.
10. 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.
11. M. Kantola, H. Mannila, K.-J. Rih, and H. Siirtola. Discovering functional and inclusion dependencies in relational databases. *International Journal of Intelligent Systems*, 12:591–607, 1992.
12. Z. Lacroix and T. Critchlow. *Bioinformatics – Managing Scientific Data*. Morgan Kaufmann Publishers, 2003.
13. H. Mannila. Methods and problems in data mining. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 41–55, 1997.
14. H. Mannila and K.-J. Raeihae. On the complexity of inferring functional dependencies. *Discrete Appl. Math.*, 40(2):237–243, 1992.
15. F. D. Marchi, S. Lopes, and J.-M. Petit. Efficient algorithms for mining inclusion dependencies. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 464–476, London, UK, 2002.
16. V. Matos and B. Grasser. SQL-based discovery of exact and approximate functional dependencies. *SIGCSE Bull.*, 36(4):58–63, 2004.

17. 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.
18. A. S. Sidhu and T. S. Dillon, editors. *Biomedical Data and Applications*, volume 224 of *Studies in Computational Intelligence*. Springer, 2009.
19. 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.

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
50	978-3-86956-144-8	**Data in Business Processes**	Andreas Meyer, Sergey Smirnov, Mathias Weske
49	978-3-86956-143-1	**Adaptive Windows for Duplicate Detection**	Uwe Draisbach, Felix Naumann, Sascha Szott, Oliver Wonneberg
48	978-3-86956-134-9	**CSOM/PL: A Virtual Machine Product Line**	Michael Haupt, Stefan Marr, Robert Hirschfeld
47	978-3-86956-130-1	**State Propagation in Abstracted Business Processes**	Sergey Smirnov, Armin Zamani Farahani, Mathias Weske
46	978-3-86956-129-5	**Proceedings of the 5th Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering**	Hrsg. von den Professoren des HPI
45	978-3-86956-128-8	**Survey on Healthcare IT systems: Standards, Regulations and Security**	Christian Neuhaus, Andreas Polze, Mohammad M. R. Chowdhuryy
44	978-3-86956-113-4	**Virtualisierung und Cloud Computing: Konzepte, Technologiestudie, Marktübersicht**	Christoph Meinel, Christian Willems, Sebastian Roschke, Maxim Schnjakin
43	978-3-86956-110-3	**SOA-Security 2010 : Symposium für Sicherheit in Service-orientierten Architekturen ; 28. / 29. Oktober 2010 am Hasso-Plattner-Institut**	Christoph Meinel, Ivonne Thomas, Robert Warschofsky et al.
42	978-3-86956-114-1	**Proceedings of the Fall 2010 Future SOC Lab Day**	Hrsg. von Christoph Meinel, Andreas Polze, Alexander Zeier et al.
41	978-3-86956-108-0	**The effect of tangible media on individuals in business process modeling: A controlled experiment**	Alexander Lübbe
40	978-3-86956-106-6	**Selected Papers of the International Workshop on Smalltalk Technologies (IWST'10)**	Hrsg. von Michael Haupt, Robert Hirschfeld
39	978-3-86956-092-2	**Dritter Deutscher IPv6 Gipfel 2010**	Hrsg. von Christoph Meinel und Harald Sack
38	978-3-86956-081-6	**Extracting Structured Information from Wikipedia Articles to Populate Infoboxes**	Dustin Lange, Christoph Böhm, Felix Naumann
37	978-3-86956-078-6	**Toward Bridging the Gap Between Formal Semantics and Implementation of Triple Graph Grammars**	Holger Giese, Stephan Hildebrandt, Leen Lambers
36	978-3-86956-065-6	**Pattern Matching for an Object-oriented and Dynamically Typed Programming Language**	Felix Geller, Robert Hirschfeld, Gilad Bracha
35	978-3-86956-054-0	**Business Process Model Abstraction : Theory and Practice**	Sergey Smirnov, Hajo A. Reijers, Thijs Nugteren, Mathias Weske
34	978-3-86956-048-9	**Efficient and exact computation of inclusion dependencies for data integration**	Jana Bauckmann, Ulf Leser, Felix Naumann

ISBN 978-3-86956-148-6
ISSN 1613-5652