Scalable Discovery of Unique Column Combinations

Arvid Heise♦[*] Jorge-Arnulfo Quiané-Ruiz★ Ziawasch Abedjan♦[*]
Anja Jentzsch♦ Felix Naumann♦[*]

♦Hasso Plattner Institute (HPI)
Postdam, Germany

★Qatar Computing Research Institute (QCRI)
Doha, Qatar

{arvid.heise, ziawasch.abedjan, anja.jentzsch, felix.naumann}@hpi.uni-potsdam.de
jquianeruiz@qf.org.qa

ABSTRACT

The discovery of all unique (and non-unique) column combinations in a given dataset is at the core of any data profiling effort. The results are useful for a large number of areas of data management, such as anomaly detection, data integration, data modeling, duplicate detection, indexing, and query optimization. However, discovering all unique and non-unique column combinations is an NP-hard problem, which in principle requires to verify an exponential number of column combinations for uniqueness on all data values. Thus, achieving efficiency and scalability in this context is a tremendous challenge by itself.

In this paper, we devise Ducc, a scalable and efficient approach to the problem of finding all unique *and* non-unique column combinations in big datasets. We first model the problem as a graph coloring problem and analyze the pruning effect of individual combinations. We then present our hybrid column-based pruning technique, which traverses the lattice in a depth-first and random walk combination. This strategy allows Ducc to typically depend on the solution set size and hence to prune large swaths of the lattice. Ducc also incorporates row-based pruning to run uniqueness checks in just few milliseconds. To achieve even higher scalability, Ducc runs on several CPU cores (scale-up) and compute nodes (scale-out) with a very low overhead. We exhaustively evaluate Ducc using three datasets (two real and one synthetic) with several millions rows and hundreds of attributes. We compare Ducc with related work: Gordian and HCA. The results show that Ducc is up to more than 2 orders of magnitude faster than Gordian and HCA (631x faster than Gordian and 398x faster than HCA). Finally, a series of scalability experiments shows the efficiency of Ducc to scale up and out.

1. INTRODUCTION

We are in a digital era where many emerging applications (e.g., from social networks to scientific domains) produce huge amounts of data that outgrow our current data processing capacities. These emerging applications produce very large datasets not only in terms of the number of rows, but also in terms of the number of columns. Thus, understanding such datasets before actually querying them is crucial for ensuring both data quality and query performance.

Data profiling is the activity of discovering and understanding relevant properties of datasets. One important task of data profiling is to discover unique column combinations (*uniques* for short) and non-unique column combinations (*non-uniques*). A unique is a set of columns whose projection has no duplicates. Knowing all uniques and non-uniques helps understand the structure and the properties of the data [1, 25]. Uniques and non-uniques are useful in several areas of data management, such as anomaly detection, data integration, data modelling, duplicate detection, indexing, and query optimization. For instance, in databases, uniques are primary key candidates. Furthermore, newly discovered uniqueness constraints can be re-used in other profiling fields, such as functional dependency detection or foreign key detection.

However, many uniques and non-uniques are unknown and hence one has to discover them. The research and industrial communities have paid relatively little attention to the problem of finding uniques and non-uniques. Perhaps this lack is due to the nature of this problem: the number of possible column combinations to be analyzed is exponential in the number of attributes. For instance, a brute-force approach would have to enumerate $2^{94} - 1$ column combinations to find all uniques and non-uniques in a dataset with 94 attributes (as one of the datasets we use in our experiments). Performing this enumeration is infeasible in practice. As a result, commercial products limit the search space to column combinations with only few columns [17, 22, 23]. This restriction loses the insights that long uniques and non-uniques (i.e., those with many attributes) can offer. For example, in bio-informatics, long uniques might lead to the detection of unknown principles between protein and illness origins [20]. Long non-uniques might lead to the detection of surprising *partial* duplicates (i.e., rows having duplicate values in many attributes) in a relation. Moreover, assessing and integrating many data sources is a widespread problem that requires fast data analysis. In the life sciences domain, integrating datasets on genes, proteins, targets, diseases, drugs, and patients helps discovering and developing drugs. The amount of publicly available data that is relevant for this task has grown significantly over the recent years [11]. Thus, scientists need new, more efficient ways to discover and understand datasets from different data sources.

Some existing work has focused on providing more efficient techniques to discover uniques [1,10,14,16,25]. For example, Gordian [25] pre-organizes the data of a relation into a prefix tree and discovers maximal non-uniques by traversing the prefix tree; Gianella and Wyss propose a technique that is based on the apriori intuition, which says that supersets of already discovered uniques and subsets of discovered non-uniques can be pruned from further analysis [10]; HCA discovers uniques based on histograms and value-

[*]Research performed while at QCRI.

Proceedings of the VLDB Endowment, Vol. 7, No. 4
Copyright 2013 VLDB Endowment 2150-8097/13/12.

counting [1]. However, none of these techniques have been designed with very large datasets in mind and hence they scale poorly with the number of rows and/or the number of columns.

Research Challenges. Discovering all (non-)uniques is a NP-hard problem [14], which makes their discovery in very large datasets quite challenging:

(1) Enumerating all column combinations is infeasible in practice, because the search space is exponential in the number of columns. Thus, one must apply effective and aggressive pruning techniques to find a solution set in reasonable time.

(2) The solution space can also be exponential: the number of uniques might reach $\binom{n}{n/2}$, where n is the number of columns. There can be no polynomial solution in such cases.

(3) Parallel and distributed approaches are a natural choice in such settings. However, they require exchanging a large number of messages among the processes to avoid redundant work. Thus, achieving scalability in this context is a tremendous challenge by itself.

Contributions and structure of this work. We address the problem of efficiently finding all uniques and non-uniques in big datasets as follows:

(1) We formally define (non-)uniques and state the problem of finding all uniques and non-uniques (Section 2).

(2) We then model unique discovery as a graph processing problem and analyze the pruning potential of individual uniques and non-uniques. We observed that aggressive pruning of the lattice (especially in a distributed environment) may lead to unreachable nodes. Thus, we propose a technique to find and remove such holes from the lattice and formally prove its correctness (Section 3).

(3) We present Ducc, an efficient and scalable system for *Discovering (non-)Unique Column Combinations* as part of the Metanome project (`www.metanome.de`). In particular, we propose a new hybrid graph traversal algorithm that combines the depth-first and random walk strategies to traverse the lattice in a bottom-up and top-down manner at the same time. We additionally incorporate row-based pruning to perform uniqueness checks in only a few milliseconds and to significantly reduce its memory footprint (Section 4).

(4) Next, we present a general inter-process communication protocol to both scale Ducc up and out. Since this communication protocol is lock-free and fully asynchronous, it allows Ducc to scale with a very low synchronisation overhead. (Section 5)

(5) We experimentally validate Ducc on real and synthetic datasets and compare it with Gordian [25] and HCA [1]. The results show the high superiority of Ducc over state-of-the-art: Ducc is up to 631x faster than Gordian and 398x faster than HCA (Section 6). We also discuss how different null value semantics impact the performance of Ducc (Section 7).

2. PROBLEM STATEMENT

We define the basic concepts of unique column combinations and proceed to state the problem we solve in this paper. Given a relation R with schema S (the set of n attributes) a *unique column combination* is a set of one or more attributes whose projection has only unique rows. In turn, a *non-unique column combination* has at least one duplicate row.

Definition 1 ((Non-)Unique Column Combination). *A column combination $K \subseteq S$ is a* unique *for R, iff* $\forall r_i, r_j \in R, i \neq j : r_i[K] \neq r_j[K]$. *All other column combinations are* non-unique.

We denote the set of (non-)unique column combinations as (n)Uc. Obviously, any superset of a unique is also unique and any subset of a non-unique is also non-unique. Thus, the most interesting uniques are those that are not supersets of other uniques. We call these uniques *minimal uniques* and denote their set mUcs. In analogy, we are interested in the *maximal non-uniques* (mnUcs). Notice that a minimal unique is not necessarily the smallest unique and vice versa for maximal non-uniques. We define a minimal unique in Definition 2 and a maximal non-unique in Definition 3.

Definition 2 (Minimal Unique Column Combination). *A unique $K \subseteq S$ is minimal iff*

$$\forall K' \subset K : (\exists r_i, r_j \in R, i \neq j : r_i[K'] = r_j[K'])$$

Definition 3 (Maximal Non-Unique Column Combination). *A non-unique $K \subseteq S$ is* maximal *iff*

$$\forall K' \supset K : (\forall r_i, r_j \in R, i \neq j : r_i[K'] \neq r_j[K'])$$

It is worth noting that in the special case of a complete duplicate row in R, the set of uniques is empty and the set of non-uniques contains one maximum non-unique with all columns.

Finding a single minimal unique is solvable in polynomial time: Simply start by checking the combination of all attributes and recursively remove attributes from combinations in a depth-first manner until all removals from a given unique render it non-unique. The number of combinations to explore is $\sum_{i=0}^{n-1}(n-i) = O(n^2)$, where n denotes the number of attributes.

However, the problem we want to solve in this paper is to discover *all* minimal uniques and *all* maximal non-uniques in a given relation. A naïve approach would have to check all 2^n-1 non-empty column combinations. Each check then involves scanning over all rows to search for duplicates. Clearly, the complexity $O(2^n * \text{rows})$ renders the approach intractable. In fact, in the worst case, there can be up to $\binom{n}{n/2} \geq 2^{\frac{n}{2}}$ minimal uniques, so the solution space is already exponential. Furthermore, Gunopulos et al. have shown that the problem of finding the number of all uniques or all minimal uniques of a given database is #P-hard [14].

3. UNIQUE OR NON-UNIQUE

We now present a novel approach to column-based pruning that significantly reduces the amount of combinations to be checked. In this section, we treat individual uniqueness checks as black boxes; we add row-based pruning in Section 4.4. We thus focus on the problem to determine whether we need to check a combination or whether we can infer the result through prior pruning: Unique or non-unique – that is the question!

3.1 Aggressively Pruning the Search Space

In this paper, we model the search space as a lattice of all column combinations. Figure 1 illustrates this lattice for the first 5 attributes from TPC-H lineitem. Each node corresponds to a column combination and nodes that are in a subset/superset relationship are connected. We use discovered uniques (green squares) and non-uniques (purple circles) to prune this lattice.

We have exemplarily marked the discovery of OL (OrderKey and Lineitem) as minimal unique (dark green square) and PSLQ (PartKey, SupplyKey, Lineitem, and Quantity) as maximal non-unique (dark purple circle). Both discoveries result into a significant pruning of the lattice leaving only 8 (white hexagons) of the 31 nodes to be checked. Again, any discovery within the eight remaining nodes leads us to further pruning.

We show a larger example with the same color coding in Figure 2, which shows the lattice created from the first eight columns

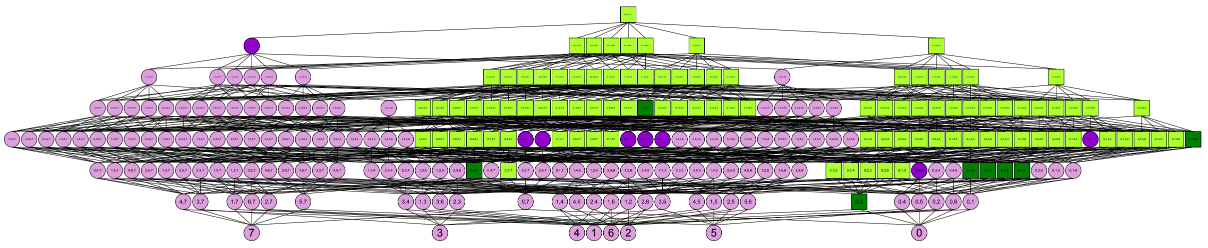


Figure 2: Pruning in eight columns of TPCH line-item (color code as in Figure 1).

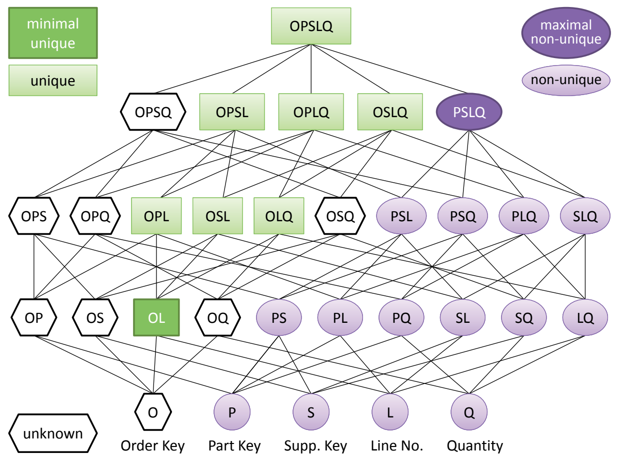


Figure 1: Effects of pruning in the lattice.

of an instance of the TPC-H lineitem table with a scale factor of 0.01. This larger example shows the complexity of the problem already for few columns and that indeed there are several uniques that one has to discover.

In general, minimal uniques at the bottom and maximal non-uniques at the top of the lattice lead to larger sets of nodes that can be pruned. Discovering a unique of size k prunes $\frac{1}{2^k}$ of the lattice. However, prior prunings might reduce the information gain: the unique AB prunes $\frac{1}{4}$ of the lattice; the unique BC additionally prunes only $\frac{1}{8}$, because ABC and its supersets are already known to be unique. Analogously, a non-unique of size k prunes $\frac{1}{2^{(n-k)}}$ of the lattice. In particular, knowing that there is a complete duplicate row and thus all columns form a non-unique already solves the problem as it prunes $\frac{1}{2^{(n-n)}}$, i.e., the entire lattice.

The greatest pruning effect can be achieved by simultaneously discovering minimal uniques and maximal non-uniques. However, one of the main challenges is to (efficiently) choose the most effective nodes to check first. Prior column-based algorithms (e.g, a-priori [10]) apply pruning, but they traverse the lattice breadth-first, which limits the effect of pruning drastically. For example in a dataset with n columns and uniques of size ≥3, a-priori with bottom-up pruning finds the first unique after $>n + \frac{n*(n-1)}{2}$ checks and prunes new combinations after $\frac{n*(n-1)*(n-2)}{6}$ more checks. Ducc immediately starts pruning at the first combination of size 3 after $n+2$ checks: Either we prune all supersets of this triple, because it is unique, or we prune all subsets, because it is non-unique. Ducc is also guaranteed to find a unique in $n + n$ checks, if one exists.

However, when applying such an aggressive pruning, some column combinations might become unreachable by lattice traversal strategies as used in Ducc, because all their super- and subsets were pruned. Thus, in the remainder of this section, we present a novel and fast technique to identify these "holes" in the lattice and prove that this technique always leads to the complete solution.

3.2 Uniques and Non-Uniques Interaction

Let us first highlight that for any given unique column combination there exists a minimal unique column combination that is a subset of or equal to the given column combination. Formally,

Lemma 1. $K \in \text{Uc} \Leftrightarrow \exists K' \in \text{mUcs} : K' \subseteq K$

Proof (Sketch). Either K is already minimal, or we can iteratively remove columns as long as K remains unique. By Definition 2, the result after this removal process is a minimal unique. The negation of the equivalence means that if we cannot find such a minimal combination, then K is non-unique. □

We remark that it suffices to know all minimal unique column combinations to classify any other column combination as either unique or non-unique. We state this in Lemma 2.

Lemma 2. *Given only the set* mUcs *of all minimal uniques and given any column combination K, we can infer whether K is unique or non-unique without performing a check on the data.*

Proof. We know from Lemma 1 that K is unique iff $\exists K' \in$ mUcs : $K' \subseteq K$. Otherwise, we know that K is non-unique by applying the negation of Lemma 1: $K \in \text{nUc} \Leftrightarrow \nexists K' \in \text{mUcs} : K' \subseteq K$. □

It is worth noticing that Lemmata 1 and 2 are analogously true given the set mnUcs of all maximal non-uniques. Therefore, taken together, either the set mUcs or the set mnUcs is sufficient to classify each node of the lattice:

Corollary 1. *Given the complete set of* mnUcs *for a given relation, one can construct the complete set* mUcs *for that relation and vice-versa. We call the constructed set the* complementary set $\text{mnUcs}^C = \text{mUcs}$.

Notice that the two sets mUcs and mnUcs do not need to have the same cardinality. An algorithm for constructing mUcs from mnUcs is given in [25], which we adapt in an incremental version.

3.3 Finding Holes

Lattice traversal algorithms terminate when all reachable nodes have been either checked or pruned. At that point, the sets mUcs and mnUcs may be incomplete, but they contain only correct combinations. We can leverage the observations of the previous section to identify possible holes in the lattice. The basic idea is to compare mUc with mnUc^C. Intuitively, we can verify whether an algorithm produces the complete and correct results using the analogy of coloring the lattice nodes according to their uniqueness. If there exists a difference, we can then conclude (from Corollary 1) that there exist holes in the lattice. In fact, the difference of such sets effectively describes at least one column combination in each existing hole. Thus, one can use the difference to start an additional traversal until the difference of mUc with mnUc^C is empty.

Assume we miss one minimal unique $K \in$ mUcs. We start coloring the lattice two times according to Corollary 1. In other words, we color all uniques in Ucs and all non-uniques in the derived Ucs^C. As we assumed that $K \notin$ mUcs, we must color K as non-unique. This is because by definition of mUc there cannot be any other minimal unique that covers K. We, then, color the lattice according to nUcs and nUcs^C. Here, we must color K as unique, because we cannot possibly find any evidence that K is non-unique. We resolve this contradiction only by assuming that Ucs and nUcs describe different lattices and thus they are not complete. Formally, Theorem 1 shows how to check whether the results are correct and complete.

Theorem 1. *Given a lattice with its corresponding sets* mUcs *and* mnUcs *and (intermediate) solution sets* $U \subseteq$ mUcs *and* $N \subseteq$ mnUcs*, then* ($U =$ mUcs $\wedge$ $N =$ mnUcs) $\Leftrightarrow$ $N^C = U$

Proof. "$\Rightarrow$": follows directly from Corollary 1. "$\Leftarrow$": we show this direction (i.e., right to left) for U only; the case for N is analogous. In particular, we show the inverse, i.e., if $U \subset$ mUcs then $U^C \neq N$: If $U \subset$ mUcs there is at least one $K \in$ mUcs $\setminus U$. We can now show that $\exists K' \supseteq K$ with $K' \in U^C$ and $K' \notin N$, thus showing $U^C \neq N$. $K' \in U^C$ is true: As $K \notin U$, K is non-unique (according to U and Lemma 2). Thus, with Lemma 1, we can find one corresponding maximal non-unique $K' \in U^C$. $K' \notin N$ is also true: Because $K \in$ mUcs, any superset $K' \in$ Ucs, but $N \cap$ Ucs $= \emptyset$. □

3.4 Removing Holes

Having shown how to identify holes in the lattice, we now show that algorithms that use this technique eventually converge to the complete solution.

Corollary 2. *Given a lattice with its corresponding sets* mUcs *and* mnUcs *and intermediate solution sets* $U \subset$ mUcs *and* $N \subseteq$ mnUcs *then for any* $K \in N^C \setminus U$*:*

$$(\exists K' \in \text{mUcs} \setminus U : K' \subseteq K) \vee (\exists K' \in \text{mnUcs} \setminus N : K' \supseteq K)$$

Each $K \in N^C \setminus U$ describes a column combination that should be a minimal unique according to N, but is not contained in U. Thus, this element is either a minimal unique and is added to U or it is a non-unique. In the latter case, it must lead to a new maximal non-unique according to Lemma 1, because it was not covered so far. In both cases, we append to U or N and hence eventually converge to the complete solution. The same holds if $N \neq$ mnUcs.

4. THE DUCC SYSTEM

Ducc is a system for finding unique and non-unique column combinations in big datasets. For simplicity, we assume for now that Ducc is a single-thread process running on a single node. We relax this assumption in Section 5, where we discuss how Ducc scales up to several CPU cores and scales out to multiple nodes.

4.1 Overview

Figure 3 illustrates the general architecture of Ducc. Ducc is composed of a Ducc *worker* (or *worker*, for short) that orchestrates the unique discovery process and a set of light-weight data structures (*position list index* (PLI), *(non-)uniques graph*, and *path trace*, explained later). Overall, Ducc first computes a PLI for each attribute in the input dataset. The PLI of a given attribute (or column combination) is a list of sets of tuple-ids having the same value for the given column (or column combination). Notice that PLIs

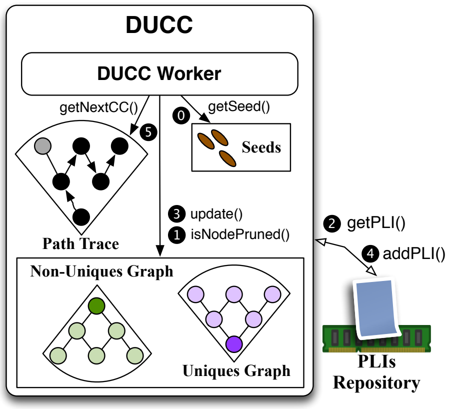


Figure 3: Ducc worker architecture.

have also been used by other researchers for discovering functional dependencies [16] and conditional functional dependencies [3].

Then, Ducc operates as follows: The Ducc worker first fetches a seed (i.e., an initial column combination) from the set of column combinations composed of two columns 0. Then, the worker consults the (non-)uniques graph to check whether it pruned the current column combination before, i.e., if it is a superset (or a subset) of an already found unique (non-unique) 1. If so, the worker then starts again from 0. Otherwise, the worker proceeds with the uniqueness check for the current column combination. For this, the worker reads the PLIs of all columns of the current column combination 2. Indeed, the worker might reuse existing PLIs of column combinations relevant to the current column combination. For instance, assume the Ducc worker has to perform the uniqueness check for the column combination ABC. If the Ducc worker had previously computed the PLI for AB, then it would intersect the PLI of AB with the PLI of C. After the uniqueness check, the worker updates the (non-)uniques graph 3. Furthermore, if the current column combination is non-unique, the worker then adds the resulting PLI to the repository 4. This repository is a main memory data structure having a least-recently-used (LRU) replacement strategy. Then, the Ducc worker fetches the next column combination to check 5 and starts again from point 1. In case that the worker does not find any unchecked column combination in the current path, it restarts from point 0. The worker repeats this process until it does not find more unchecked column combinations and seeds.

In the remainder of this section, we explain the Ducc worker in more detail (Section 4.2), then the strategies used by Ducc to efficiently traverse the lattice of column combinations (Section 4.3), and the set of light-weight data structures used by Ducc to perform fast uniqueness checks (Section 4.4).

4.2 DUCC Worker

Algorithm 1 details the way the Ducc worker operates to find all uniques and non-uniques in a given dataset. Notice that the approach followed by the Ducc worker is suitable for any bottom-up and top-down lattice-traversal strategy. The algorithm first checks each column individually for uniqueness by any suitable technique (Line 1), e.g., distinctness check in a DBMS. The worker adds all found unique columns to the set mUc (Line 2). At the same time, the worker enumerates all pairs of non-unique columns as seeds, i.e., as starting points for the graph traversal (Line 3). In the main part of the algorithm (Lines 6–14), the worker processes all seeds that are given as starting points. For this, the worker first chooses a column combination K from the seeds to check for uniqueness (Line 7). Notice that one can provide any *strategyTake*

Algorithm 1: Ducc Worker

```
Data: columns
Result: mUcs, mnUcs
1  check each column for uniqueness;
2  mUcs ← all unique columns;
3  seeds ← pairs of non-unique columns;
4  mnUcs ← $\emptyset$;
5  repeat
6      while seeds $\neq \emptyset$ do
7          K ← strategyTake(seeds);
8          repeat
9              if K is unique ∧ all subsets are known to be
               non-unique then
10                 add K to mUcs;
11             else if K is non-unique ∧ all supersets are known
               to be unique then
12                 add K to mnUcs;
13             K ← strategyStep(K);
14         until K $\neq$ null ;
15     seeds ← strategyNextSeeds();
16 until seeds = $\emptyset$ ;
```

function to decide how the worker chooses K. The advantage of providing this function is that one can control the parallelization of the Ducc process (see Section 5 for details). If K is unique and the worker already classified all subsets as non-unique, the worker then adds K to mUc (Line 10). Analogously, if K is non-unique and the worker already classified all supersets as unique, the worker then adds K to mnUc (Line 12). Next, the worker invokes the *strategyStep* function to decide on the next column combination to check for uniqueness. Notice that by providing their own *strategyStep*, users can control the way the worker driver has to traverse the graph. If there are no more column combinations to check in the current path (i.e., supersets of the current seed), the *strategyStep* function then returns null (Line 14). In this case, the worker proceeds with the next seed (Line 6). The worker repeats this main process until there are no more seeds.

However, the worker might not cover the complete lattice, because one might provide highly aggressive pruning strategies in the *strategyStep* function (see Section 3.1). Therefore, the worker may calculate a new set of seeds and reiterate the main process (Line 15). For our traversal strategies, Ducc uses the approach described in Section 3.3 to identify such possible "holes" in the lattice and use them as new seeds.

4.3 Graph Traversal Strategies

It is worth noting that the *strategyTake* and *strategyStep* functions play an important role in the performance of Ducc, because they guide the Ducc worker in how to explore the lattice (see Section 4.2). In this section, we provide two advanced graph traversal strategies (*Greedy* and *Random Walk*) that allow Ducc to traverse the lattice efficiently. These strategies quickly approach the border between uniques and non-uniques (see Section 3) and hence allow Ducc to cover the lattice by visiting only a very small number of column combinations.

Generally speaking, the main goal of the Greedy strategy is to find minimal uniques as fast as possible in order to prune all supersets from the search space. As a side effect, this strategy also prunes all subsets of non-uniques that are discovered in the process. However, a limitation of the Greedy strategy is that it is not well suited for parallel computation, because all computation units can quickly converge to the same combinations causing much redundant computation. One might think of frequently sharing local decisions among Ducc workers to deal with this issue, but this would require significant coordination among Ducc workers. This is why we introduce the Random Walk strategy, which achieves an efficient parallelization due to its random nature. Another difference is that while Greedy approaches the border between uniques and non-uniques strictly from below, Random Walk jumps back and forth over the border to approximate its shape faster. Indeed, as both strategies are two aggressive pruning techniques, they might miss some column combinations in the lattice (see Section 3.3). We now discuss in detail the two advanced graph traversal strategies.

Algorithm 2: greedyStep()

```
Data: Column combination $K_i$
Result: Next column combination $K_{i+1}$
1  if $K_i$ is non-unique then
2      remove all subsets from queue;
3      calculate estimates of unchecked supersets;
4      update queue with new estimates;
5  else
6      remove all supersets from queue;
7  $K_{i+1}$ ← take head from queue;
```

Greedy. The main idea of the Greedy strategy is to first visit those column combinations that are more likely to be unique. For this, Greedy maintains a priority queue to store distinctness estimates for potentially unique column combinations. The distinctness $d : S \to (0; 1]$ is the ratio of the number of distinct values over the number of all values. A distinctness $d(K) = 1$ means that column combination K is unique; the distinctness of a column combination with many duplicates approaches 0. A sophisticated estimation function may allow Ducc to better prune the search space, but it can be too costly to perform, outweighing any accuracy gain. Thus, we favor a simple estimation function $\tilde{d}$ inspired by the addition-law of probability:

$$\tilde{d}(P_1 P_2) = d(P_1) + d(P_2) - d(P_1) * d(P_2) \quad (1)$$

Using the above estimation function, we use a *greedyTake* function (which is an implementation of *strategyTake*) that chooses the seed with the highest estimate. Similarly, we use a *greedyStep* function (which is an implementation of *strategyStep*) as shown in Algorithm 2. For each given non-unique column combination, *greedyStep* removes all subsets of the given column combination from the priority queue (Line 2), calculates the estimates for every unchecked superset (Line 3), and updates the priority queue (Line 4). In turn, for each given unique column combination, *greedyStep* simply removes all supersets of the given column combination from the priority queue (Line 6). At the end, *greedyStep* returns the elements with the highest estimated distinctness (Line 7). This means that Greedy returns either a superset or subset of the given column combination.

Random Walk. This strategy traverses the lattice in a randomized manner to reduce both unnecessary computation and the coordination among workers. Random Walk strategy starts walking from a seed upwards in the lattice until it finds a unique and then it goes downwards in the lattice until it finds a non-unique. When Random Walk finds a non-unique, it again walks upwards looking for a unique and so on. The main idea behind Random Walk is to quickly converge to the "border" in the lattice that separates the uniques from the non-uniques and walk along such a border. All minimal uniques and maximal non-uniques lie on this border. In contrast to

Algorithm 3: randomWalkStep()

```
Data: Column combination $K_i$
Result: Next column combination $K_{i+1}$
1  push $K_i$ into trace;
2  if $K_i$ is unique $\wedge \exists$ unchecked subsets then
3      $K_{i+1} \leftarrow$ random unchecked subset;
4  else if $K_i$ is non-unique $\wedge \exists$ unchecked supersets then
5      $K_{i+1} \leftarrow$ random unchecked superset;
6  else
7      $K_{i+1} \leftarrow$ pop trace
```

Greedy, Random Walk reduces the likelihood of converging to the same column combinations when running in parallel.

Random Walk implements a *randomWalkTake* function (i.e., *strategyTake*) that chooses a seed randomly. Additionally, Random Walk maintains a *Path trace*, which is initialized with the seed. For walking through the lattice, this strategy implements a *randomWalkStep* function (i.e., *strategyStep*) that works as shown in Algorithm 3. First, Random Walk pushes the current column combination into the path trace. Next, it verifies whether the current column combination is unique. If it is, Random Walk then goes down to a random, yet unchecked subset (Line 3). If not, it analogously chooses a random superset (Line 5). If Random Walk cannot make any additional step from the current combination, it then backs up one step and uses the previous combination (Line 7). The strategy repeats this process until it completely explored the reachable lattice from the given seed. It is worth noting that Lines 3 and 4 check not only for (non-)uniqueness, but also check whether the current column combination is covered by a known unique or non-unique. This strategy allows Random Walk to lazily prune the search space from the bottom and the top similar to Greedy.

4.4 Light-Weight Data Structures

At its core, Ducc uses a set of data structures that allows it to quickly check if a given column combination is either unique or non-unique. The three most important data structures used by Ducc are: a *position list index*, a *(non-)uniques graph*, and a *path trace*. Generally speaking, Ducc uses: (i) the position list index to efficiently perform a uniqueness check, (ii) the (non-)unique graph to avoid uniqueness checks of already pruned column combinations, and (iii) the path trace to quickly obtain the next column combination (in the current path) to check. We discuss each of these data structures in the following.

Position list index. Ducc combines row-based pruning with the column-based pruning presented in Section 3. When performing a uniqueness check, we could naïvely scan over all rows until we find a duplicate if existent. However, non-uniques near the border to uniqueness usually have very few duplicates. We observed that more than 95% of the maximal non-uniques in fact have only up to 10 duplicate pairs in our real datasets with millions of rows.

The position list index (PLI) and its novel intersection algorithm is the core of the row-based pruning of Ducc. PLI is a data structure that keeps track of duplicate tuples for a specific column (or column combination). In other words, each entry in the PLI of a given column is a set of tuple-ids having the same value for the given column. For example, given an attribute a with two sets of duplicates (records r_{42} and r_{10} for the value v_1 and records r_7 and r_{23} for the value v_2), the PLI of a is as follows: $\text{PLI}_a = \{\{r_{42}, r_{10}\}, \{r_7, r_{23}\}\}$. Since PLIs track only duplicates in a column (or column combination), Ducc maintains one PLI for each non-unique column (and column combination) in a relation. Therefore, columns or column

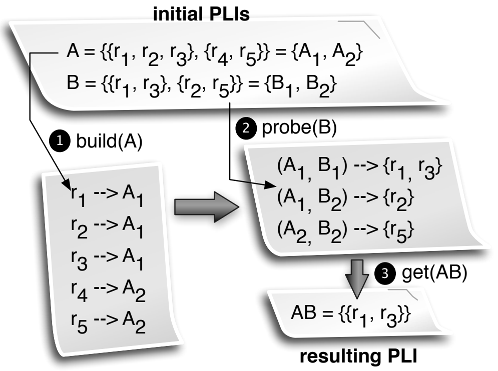


Figure 4: Example of intersecting two PLIs.

combinations without a PLI are uniques. Notice that Ducc computes the PLI of a column combination by intersecting the PLIs of its subsets. As a result of using PLIs, Ducc can also apply row-based pruning, because the total number of positions decreases monotonously with the size of column combinations. Intuitively, combining columns makes the contained combination values more specific with the tendency towards distinctness. This means that each intersection of PLIs results in smaller- or equal-size PLIs for the values of the considered column combinations. We observed that the size of such PLIs follow a power law distribution, where the size of most PLIs are in the order of KBs for TBs-sized datasets. It is worth noticing that Ducc intersects two PLIs in linear time in the size of the smaller PLI. This allows Ducc to perform an intersection in a few milliseconds. In fact, Ducc intersects two PLIs in a similar way in which a hash join operator would join two relations.

Figure 4 shows an example of intersecting two PLIs (A and B). The PLIs A and B are composed of two different sets of duplicate records each: A_1 and A_2 for A and B_1 and B_2 for B. In this scenario, Ducc first builds a mapping between each duplicate record r_i to the set of duplicates they point to ❶, e.g., r_1 points to set A_1. Ducc uses this mapping to probe each duplicate record in B ❷. This results in a set of duplicate records that appear in both PLIs. For example, records r_1 and r_3 appear in the resulting sets A_1 and B_1, record r_2 appears in the resulting sets A_1 and B_2, and record r_5 appears in the resulting sets A_2 and B_2. Notice that record r_4 does not appear in any resulting set, because it appears in only one set (set A_2). Finally, Ducc keeps those resulting sets with more than one record ❸. In this example, Ducc retains the set with the records r_1 and r_3.

(Non-)Uniques graph. The (non-)unique graph is a data structure that maintains, for each column, a list of non-redundant uniques and non-uniques containing the column. This data structure allows Ducc to use a lazy pruning strategy rather than using an eager pruning strategy[1]. In particular, Ducc uses the (non-)uniques graph to check if any given column combination is a superset (or subset) of an already found unique (non-unique) column combination. The main goal of performing this check for a given column combination is to save CPU cycles by avoiding intersecting the PLIs of the columns in the given column combination as well as ending the search path as early as possible. The challenge is that we have to check whether a superset or subset of any given column combination exists in the order of few milliseconds. Otherwise, this checking becomes more expensive than the PLIs intersection itself. For each column, Ducc indexes all non-redundant (non-)uniques in which the column is involved in order to achieve fast lookups of the

[1]Using an eager pruning strategy would require Ducc to materialize the entire lattice in main memory, which is infeasible.

(non-)uniques graph. Indeed, the index size of a column might become too large, which, in turn, increases lookup times. Therefore, we use a main memory-based, dynamic hash-based index structure. The idea is that whenever the index of a given column becomes too big, we split the index into smaller indexes of column combinations having such a column. For example, assume a relation with four attributes: A, B, C, and D. If the index for column A (or for column combination BD) goes beyond a given threshold, we split such an index into the indices AB, AC, and AD (respectively, into indexes ABD, BCD). This allows us to guarantee on average fast lookups on the (non-)uniques graph.

Path trace. Since DUCC can traverse the lattice by going up and down, DUCC uses this data structure to efficiently find another path to explore when it finishes checking all column combinations of a single path. The way DUCC implements this data structure depends on the graph traversal strategy it uses. Random Walk keeps track of previously visited column combinations in a stack-like trace. In contrast, Greedy maintains a Fibonacci heap that ranks column combinations by their estimated distinctness.

5. SCALING DUCC UP AND OUT

So far, we assumed that DUCC runs on a single computing node and without multi-threading. While that setup might be enough for several applications, when dealing with big data one should use several CPU cores and multiple computing nodes. We now relax the assumption and discuss how DUCC can scale to several CPU cores as well as to several computing nodes. In this section, we present a general inter-process communication protocol to scale DUCC up and out at the same time with a low synchronisation overhead. We discuss these two points in the remainder of this section.

5.1 Scale Up

As DUCC is mainly CPU-bound, one might think that by scaling the CPU up (in terms of speed and number of cores) DUCC can achieve linear scalability by running in a multi-threading manner. However, multi-threading comes at a price: it usually incurs a high overhead due to data structure locking and threads coordination [26]. To overcome this issue, DUCC mainly relies on a *lock-free coordination* mechanism, which allows DUCC to have almost no overhead when scaling up to several threads.

Lock-free worker coordination. Running on multiple workers requires DUCC to propagate the observations[2] done by each worker to others workers in order to avoid redundant computations across workers. A simple way of doing this would be by sharing the (non-)uniques graph among all DUCC workers. DUCC would require a locking mechanism to coordinate all write operations to these graphs. However, it has been shown by other researchers that locking mechanisms usually incur high overheads [26]. Therefore, DUCC uses a lock-free coordination mechanism to propagate observations among workers. This mechanism mainly relies on two features. First, each DUCC worker maintains a local copy of all uniques and non-uniques column combinations already observed by other workers (*internal (non-)uniques graph*, for short). Second, DUCC workers share a *local event bus* to efficiently propagate observations across workers. Thus, the synchronization between threads is reduced to concurrent access on the event bus. Figure 5 illustrates the distributed architecture of DUCC.

[2]An observation is a mapping of a column combination to a status type. There exist six different possible statuses for a column combination: MUC, MUC candidate, UC, MNUC, MNUC candidate, and NUC.

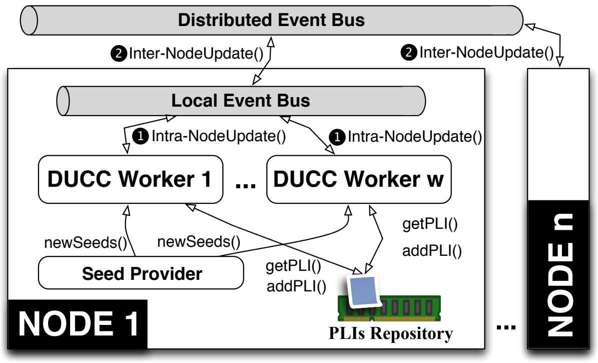


Figure 5: DUCC distributed architecture.

Producer-consumer pattern. This local event bus operates in a *producer-consumer* manner to avoid that a DUCC worker waits for observations made by other DUCC workers. Each DUCC worker subscribes and exposes an event queue to the local event bus. When a DUCC worker finishes with the uniqueness check of a given column combination, it updates its internal (non-)uniques graph with the resulting observation, and pushes such an observation to the local event bus (❶ in Figure 5). In turn, the local event bus enqueues every incoming observation into the event queue of each subscribed DUCC worker. Then, a DUCC worker updates its internal (non-)uniques graph with the observations that are in its own event queue, i.e., with the observations made so far by other workers. DUCC workers pull observations from their queues right after pushing their own observations to the local event bus. The main advantage of this mechanism is that it allows a DUCC worker to update its internal (non-)uniques graph as well as to push and pull the resulting observations in the order of microseconds.

Asynchronous seed provider. A limitation in scaling up is the discovery of holes. Near the end of one iteration of DUCC (i.e., when all workers eventually run out of seeds), workers redundantly perform the same calculation to find new seeds. Therefore, we extracted the seed calculation process into a separate thread "seed provider". The seed provider continuously tries to detect new holes when new minimal uniques or maximal non-uniques have been found and propagated over the local event bus. The order in which DUCC chooses each seed depends on the strategy used by DUCC to traverse the graph (in particular on the *strategyTake* function).

5.2 Scale Out

Indeed, only scaling DUCC up does not help us to deal with big datasets in an efficient manner, as big datasets are typically in the order of terabytes or petabytes: bringing one petabyte of raw data into main memory using a single computing node with one hard disk (having a sustained rate of 210MB/s) would take 59 days. Therefore, it is crucial for DUCC to scale out to many computing nodes to efficiently deal with big datasets. To achieve this, DUCC uses the Hadoop MapReduce framework to parallelize the unique discovery process across several computing nodes. While DUCC executes one MapReduce job to create the initial PLIs and distribute it with HDFS, the main lattice traversal runs in a *map-only* MapReduce job where each map task takes a seed at random and traverse the graph starting from the chosen seed as explained in Section 4. To prune the search space, each map task maintains a PLI, a (Non-)Uniques Graph, and a Path Trace data structure locally, as explained in Section 4.4. However, maintaining these three light-weight data structures only locally would make each map task perform redundant work. Thus, map tasks share their observations with each other. However, propagating hundreds of thousands (or even millions) of observations through the network would also negatively impact the performance of Ducc.

To deal with this problem, Ducc uses a *selective inter-node communication*, which allows Ducc to make a tradeoff between network traffic and redundant work. The idea is to propagate only the observations concerning minimal uniques and maximal non-uniques across nodes (4 in Figure 5), which by definition prune more combinations than non-minimal/maximal combinations and we have thus a good ratio between communication overhead and pruning effect. Additionally, each worker needs less time to maintain its (non-)unique graph, which becomes increasingly important when scaling out to multiple of computing nodes. Ducc leverages Kafka (`kafka.apache.org`) as *distributed event bus* to efficiently propagate local observations to all Ducc workers. Similar to the local event bus (see Section 5.1), the distributed event bus follows the same producer-consumer pattern. The local event bus subscribes and exposes an event queue to the the distribute event bus, which in turn enqueues every incoming observation into the queue of each subscriber. Nonetheless, in contrast to the local event bus, the distributed event bus propagates observations regarding only minimal uniques and maximal non-uniques to avoid congesting the network.

6. EXPERIMENTS

We evaluate the efficiency of Ducc to find minimal unique column combinations and maximal non-unique column combinations. We compare Ducc with two state-of-the-art approaches: Gordian [25] and HCA [1]. We perform the experiments with three main objectives in mind: (i) to evaluate how well Ducc performs with different numbers of columns and rows in comparison to related work; (ii) to measure the performance of Ducc when *scaling up* to several CPU cores; (iii) to study how well Ducc *scales out* to several computing nodes.

6.1 Setup

Server. For all our single node experiments, we use a server with two 2.67GHz Quad Core Xeon processors; 32GB of main memory; 320GB SATA hard disk; Linux CentOS 5.8 64-bit; 64-bits Java 7.0.

Cluster. For our scale-out experiments, we use a cluster of nine computing nodes where each node has: Xeon E5-2620 2GHz with 6 cores; 24GB of main memory; 2x 1TB SATA hard disk; one Gigabit network card; Linux Ubuntu 12.04 64-bit version. Each computing node has a 64-bits Java 7.0 version installed. One node acts as a dedicated Hadoop, Kafka, and ZooKeeper master.

Datasets. We use two real-world datasets and one synthetic dataset in our experiments. The North Carolina Voter Registration Statistics (NCVoter) dataset contains non-confidential data about 7,503,575 voters from the state of North Carolina. This dataset is composed of 94 columns and has a total size of 4.1GB. The Universal Protein Resource (UniProt, `www.uniprot.org`) dataset is a public database of protein sequences and functions. UniProt contains 539,165 fully manually annotated curated records and 223 columns, and has a total size of 1GB. Additionally, we use the synthetic *lineitem* table with scale-factor 1 from the TPC-H Benchmark. The *lineitem* table has 16 columns. For all datasets the number of unique values per column approximately follows a Zipfian distribution: few columns have very many unique values and most columns have very few unique values.

Systems. We use Gordian [25] and HCA [1] as baselines. While Gordian is a row-based unique discovery technique, HCA is an improved version of the bottom-up apriori technique presented in [10]. We made a best-effort java implementation of Gordian according to the description given in [25]. For HCA, we use the same prototype as in [1], but, for fairness reasons, we store the input dataset in main memory rather than in a disk-based database. For the scale-out experiments, we use Hadoop v0.20.205 with the default settings. For all our experiments, we execute these three systems three times and report the average execution times. Finally, it is worth noting that we do not show the results for Ducc using the Greedy strategy, because the results are very similar to the results obtained by Ducc when using the Random Walk strategy.

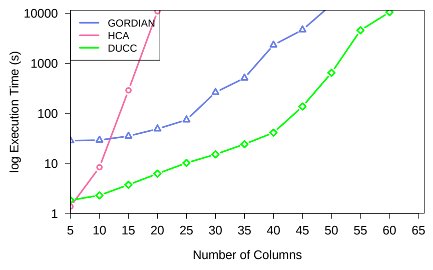
6.2 Scaling the Number of Columns

In these experiments, we vary the number of columns to evaluate how well Ducc performs with respect to wide tables. We limit the number of rows to 100*k* to better evaluate the impact in performance of having a different number of columns.

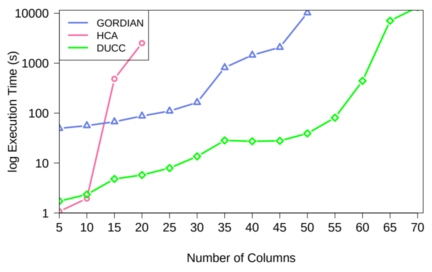
Figure 6a illustrates the results for NCVoter. For few columns, such as 5 and 10, all algorithms finish in seconds. However, in both cases Gordian performs worst by needing each time more than 20 seconds while Ducc and HCA both finish within 2 seconds for 5 columns. On the dataset with 10 columns. Ducc is with 3 seconds runtime already faster than HCA, which needs more than 8 seconds. From 15 columns, Gordian starts to outperform HCA by nearly one order of magnitude. As the number of columns increases the bottom-up approach HCA runs into problems. The runtime of Ducc stays below 4 seconds, outperforming HCA by 2 orders of magnitude and Gordian by one order of magnitude. On the dataset with 20 columns HCA is already by more than 2 orders of magnitudes slower than Gordian and is not able to finish on the dataset with 25 columns within 10 hours. Comparing to HCA, the runtime of Gordian increases moderately until 55 columns when Gordian is also not able to finish in 10 hours. On 50 columns Ducc is still one order of magnitude faster than Gordian and is also able to process the NCVoter dataset with up to 65 columns in 5.7 hours.

Figure 6b shows the results for the UniProt dataset. In these results, we observe a similar behaviour of all three systems as in the results for the NCVoter dataset. As this dataset has fewer and smaller uniques, all algorithms perform better than on the NCVoter dataset. In the experiments with 5 and 10 columns, Ducc is slightly slower than HCA since the only existent uniques are single columns, which benefit HCA. Still, Ducc is one order of magnitude faster than Gordian, which mainly suffers from the overhead of creating the prefix tree. From 15 columns on, we observe that the runtime behavior is similar to the experiments on the NCVoter dataset. Ducc significantly outperforms HCA and increases its improvement factor over Gordian as more and larger uniques can be found in the lattice. For example, Ducc is already two orders of magnitude faster than HCA for 15 columns: Ducc runs in 4.7 seconds while HCA runs in 485 seconds. This difference in performance increases significantly on 20 columns, where Ducc runs in 5.7 seconds and HCA runs in 2,523 seconds. Again, we aborted HCA after 10 hours for the experiment with 25 columns. Ducc outperforms Gordian already from 25 columns on by more than one order of magnitude. When running over 50 columns, Ducc is almost three orders of magnitude faster than Gordian. We also observe that only Ducc finishes the experiment on 70 columns (in nearly three hours).

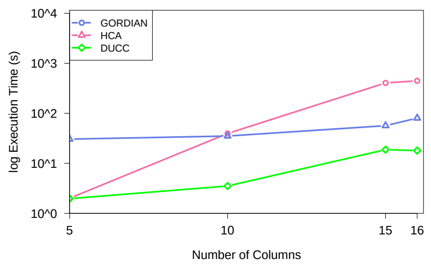
Figure 6c illustrates the results for the lineitem table from the TPC-H benchmark. These results show again that, for few columns (5 columns), Ducc has the same performance as HCA and more than one order of magnitude better performance than Gordian. Ducc and HCA finished both in 2 seconds while Gordian needed 30 seconds. Ducc significantly outperforms both Gordian and HCA on more than 5 attributes. In general, as lineitem contains



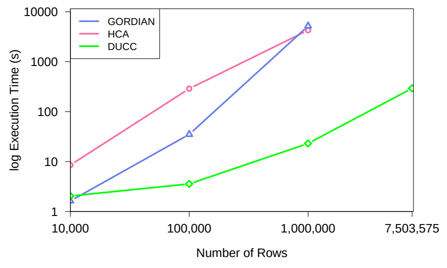
(a) NCVoter



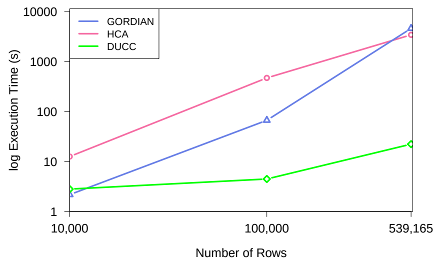
(b) UniProt



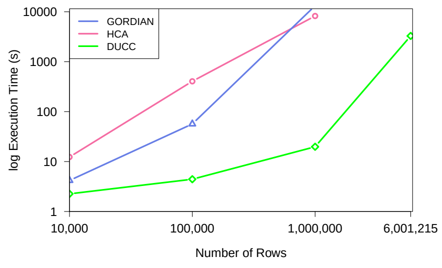
(c) TPC-H



(d) NCVoter



(e) UniProt



(f) TPC-H

Figure 6: Scaling the number of columns on 100,000 rows (top) and scaling the number of rows on 15 columns (bottom).

only 16 columns, all algorithms could deal with the column dimension of this dataset. However, Ducc was one order of magnitude faster than HCA running on 10 columns and two orders of magnitude on 15 and 16 columns. Throughout all column configurations, Ducc was one order of magnitude faster than Gordian.

In summary, we observe that HCA has low performance on datasets with many columns: HCA must verify all column combinations on the lower levels of the lattice in order to discover minimal uniques of large size. In contrast, Ducc keeps visiting relevant nodes around the minimal uniques and maximal non-uniques border only. For example, on the UniProt dataset with 20 columns and 92 minimal uniques in a solution space of $2^{20} - 1 = 1,048,575$ combinations, Ducc performs only 756 uniqueness checks resulting into 1,156 intersections while HCA has to perform 31,443 verifications. Furthermore, Gordian runs into performance bottlenecks at two stages. First, Gordian mainly operates on a prefix tree, which is expensive to create in many cases. Second, Gordian requires considerable amount of time to generate minimal uniques from maximal non-uniques when input datasets have a high number of minimal uniques. Finally, we observe that, in contrast to Gordian and HCA, Ducc mainly depends on the number of minimal uniques and not on the number attributes.

6.3 Scaling the Number of Rows

We now evaluate Ducc under a different number of rows and compare it to both Gordian and HCA. For this, we fixed the number of columns of each dataset at 15, because Gordian and HCA significantly decrease their performance for more columns. Then, we scale each dataset (starting from 10,000 rows) by a factor of 10 until we reached the total size of the dataset.

Figure 6 shows the results for these experiments. We observe in Figure 6d that only Ducc was able to finish the total dataset within the 10 hour time frame: it finishes in 294 seconds. We had to abort both Gordian and HCA after ten hours. We observe that Ducc outperforms both Gordian and HCA in general. In particular, we observe that the improvement factor of Ducc over both Gordian and HCA increases as the number of rows increases. For example, Ducc is 5x faster than HCA (and 1.25x faster than Gordian) for 10k rows and 185x faster than HCA (225x faster than for Gordian) for 1 million rows.

Figure 6e illustrates the results for UniProt, which is much smaller than NCVoter. We observe that Ducc outperforms both Gordian and HCA from 100,000 rows by more than one order of magnitude. Especially, Ducc outperforms both systems by two orders of magnitude on the complete dataset.

Figure 6f shows the results for TPC-H. Here, we observe the same behavior as for NCVoter and UniProt. Ducc outperforms Gordian and HCA by more than two orders of magnitude. In fact, Ducc is the only one that was able to finish line item with a scale factor of 1 (which contains more than 6 million rows).

In general, we see that Ducc is clearly superior to both baseline systems in all our experiments. Especially, we observed that this is due to the fact that Ducc mainly depends on the number of minimal uniques and not on the number of columns. We study this aspect in detail in the next subsection. Notice that we focus only on Ducc in these new experiments since we already showed that it is much faster than both baseline systems.

6.4 Number of Uniqueness Checks

In the previous two subsections, we observed that Ducc performs increasingly better on larger numbers of columns and rows than the previous approaches. The reason is that, in contrast to state-of-the-art algorithms, Ducc mainly depends on the solution set size and not on the number of columns. As a result, Ducc conducts a much smaller number of uniqueness checks.

Figure 7 illustrates the correlation between the number of uniqueness checks performed by Ducc and the number of minimal uniques for NCVoter with 5 to 70 columns. In this figure, the line denotes the regression line and the data points are the real number of uniqueness checks performed by Ducc to find all minimal uniques. Please note that the points for up to 35 columns are near to (0, 0). We observe a strong correlation (coefficient of determination $\mathbf{R}^2$=0.9983) between the number of checks performed by Ducc and the number of minimal uniques. This clearly shows that

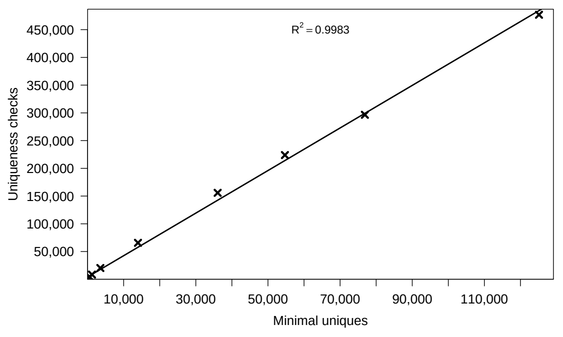


Figure 7: Correlation analysis of mUcs and Uniqueness checks.

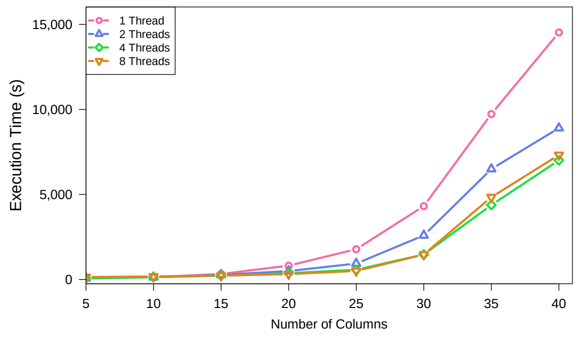


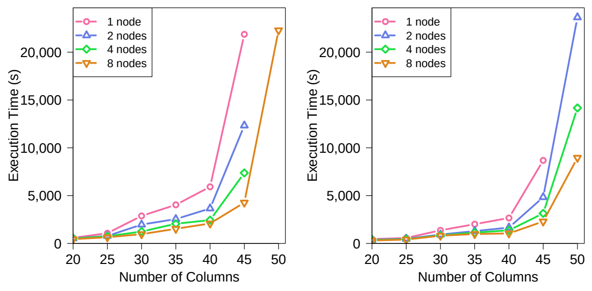
Figure 8: Scale-up on the entire NCVoter dataset.

Ducc approaches the border between uniques and non-uniques very efficiently. For example for 70 columns, Ducc performs 476,881 intersections to find the 125,144 minimal uniques. This number of checks is roughly double as high as the lower bound to find all minimal uniques. The lower bound, in turn, is given by the number of minimal uniques plus the distinct number of their subsets to verify minimality. It is worth noting that Ducc performs more checks than the lower bound, because it has to check some additional column combinations that are also candidates for being in mUc or mnUc.

6.5 Scale-Up

So far, we could observe the high superiority of Ducc over state-of-the-art systems. Noneteheless, we also observed in Figures 6a and 6b that Ducc could not finish within 10 hours. Therefore, we now evaluate how well Ducc exploits multiple CPU cores to improve execution times. For this, we use the entire NCVoter dataset since this is the biggest dataset and Ducc showed its limitations with this dataset. Later, in Section 6.6, we study how well Ducc scales to multiple compute nodes to further improve performance.

Figure 8 shows the scale up results for Ducc. As expected, we see that using more workers speed up Ducc significantly with increasing number of columns. In particular, we observe that four working threads perform more than twice faster than one thread for 20 or more columns. Adding the hyperthreaded cores does not speed up the task any further. The suboptimal speedup is caused by overlapping paths of the random walk algorithm. Although redundant intersections can be avoided with our (non-)unique graphs, Ducc still needs to find unseen paths for each thread. This becomes increasingly difficult with later iterations. Especially, plugging holes becomes a bottleneck for parallelization, because the seed provider takes longer to detect the holes than the worker threads need to plug them. It is worth noticing that, at the end, many of the working threads idle or perform redundant checks in parallel.



(a) Nodes with 1 thread (b) Nodes with 4 threads

Figure 9: Scale-Out on the entire NCVoter dataset

In summary, adding more working threads speeds up Ducc significantly, especially in the beginning. However, scaling up does not help to process datasets that do not fit into main memory.

6.6 Scale-Out

In Sections 6.2 and 6.3, we have considered a single node in all our experiments for fairness reasons with respect to baseline systems. We now release this assumption and run Ducc on up to eight nodes to measure the impact of scaling out. At the same time, we also scale-up on each individual node.

Figure 9 shows the results when using 1, 2, 4, and 8 computing nodes (each node with 1 and 4 threads). We see a similar speed-up when scaling out in comparison to scaling up. Interestingly, there is not a significant difference in running 4 workers on 1 machine or 4 machines with 1 worker. Thus, depending on the infrastructure at hand, users may decide to speed up the process by using a few big machines or several small machines. For example, for 45 columns, increasing the nodes with one thread from two to eight decreases the runtime by factor 3. Increasing the number of threads to four further halves the runtime. Finally, we also observed in our experiments that the overhead of scaling out with Hadoop is negligible: 1.5 minutes on average. These results show the high efficiency of Ducc to scale out at a very low overhead.

7. NULL-VALUE SEMANTICS

In our discussion and problem definition we have yet ignored the presence of null-values (⊥) in the data. We have thus implicitly assumed a SQL-semantic in which (⊥ = ⊥) evaluates to `unknown` and thus two null-values do not constitute a duplication of values. Discovered uniques conform to SQL's `UNIQUE` constraint, which allows only unique non-⊥-values and multiple ⊥-values. In an extreme case, a column with only nulls forms a minimal unique.

Table 1: Example relation for different null-values semantics.

A	B	C	D
a	1	x	1
b	2	y	2
c	3	z	5
d	3	⊥	5
e	⊥	⊥	5

An alternative semantics is to let (⊥ = ⊥) evaluate to `true`: in effect ⊥-values are not distinct and multiple ⊥-values render a column non-unique. To illustrate the difference for our problem, regard the small relation of Table 1. Under SQL semantics we can identify both A and C as minimal uniques and BD as maximal non-unique. Maybe surprisingly and for a data analyst un-intuitive, the

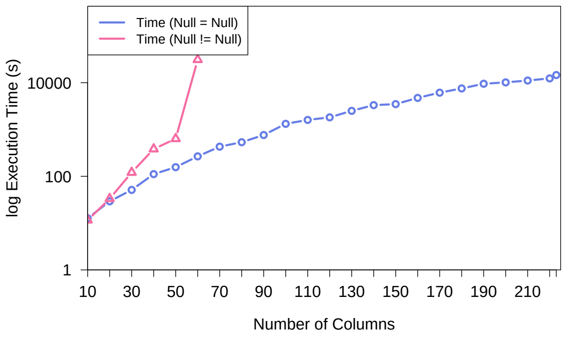


Figure 10: Comparing the null semantics in UniProt.

uniqueness of C implies that also CD is unique. This interpretation changes under the alternative semantics. There, A and BC are minimal uniques, and BD and CD are maximal non-uniques. Both semantics are of interest in a data profiling context.

The implementation of this alternative requires only a small change in the Ducc algorithm: When creating the initial PLIs, we retain a group of all ⊥-values and thus mark them as duplicates. In this way, we save significant time, because the solution set contains orders of magnitudes fewer minimal uniques. Hence, Ducc can process the complete lattice more quickly.

Figure 10 shows an experiment in which we were able to detect all 841 minimal uniques in the complete UniProt dataset with 223 columns in only 4 hours. Notice that this is infeasible with state-of-the-art algorithms even with this alternative semantics of null-values. Furthermore, the alternative semantic has the additional benefit that both the number as well as the size of the minimal uniques are more manageable: The largest minimal unique had nine columns and the median minimal unique seven. With the SQL semantics, minimal uniques can easily consist of half the columns.

8. RELATED WORK

Even if the topic of discovering unique column combinations is of fundamental relevance in many fields (such as databases and bioinformatics), there have been only few techniques to solve this problem. Basically, there exist only two different classes of techniques in the literature: column-based and row-based techniques [1, 10, 25].

Row-based techniques benefit from the intuition that non-uniques can be detected without considering all rows in a table. Gordian [25] is an example of row-based techniques. Gordian pre-organizes the data of a table in form of a prefix tree and discovers maximal non-uniques by traversing the prefix tree. Then, Gordian computes minimal uniques from maximal non-uniques. The main drawback of Gordian is that it requires the prefix tree to be in main memory. However, this is not always possible, because the prefix tree can be as large as the input table. Furthermore, generating minimal uniques from maximal non-uniques can be a serious bottleneck when the number of maximal non-uniques is large [1].

Column-based techniques, in turn, generate all relevant column combinations of a certain size and verify those at once. Giannella et al. proposed a column-based technique for unique discovery that can run bottom-up, top-down, and hybrid with regard to the powerset lattice of a relation's attributes [10]. Their proposed technique is based on the apriori intuition that supersets of already discovered uniques and subsets of discovered non-uniques can be pruned from further analysis [2]. However, this approach does not scale in the number of columns, as realistic datasets can contain uniques of very different size among the powerset lattice of column combinations. Furthermore, their verification step is costly as it does not use any row-based optimization. In [1], we presented HCA, an improved version of the bottom-up apriori technique presented in [10]. HCA performs an optimized candidate generation strategy, applies statistical pruning using value histograms, and considers functional dependencies (FDs) that have been inferred on the fly. Furthermore, we combined the maximal non-unique discovery part of Gordian with HCA, leading to some performance improvements on datasets with large numbers of uniques. However, as HCA is based on histograms and value-counting, there is no optimization with regard to early identification of non-uniques in a row-based manner.

Ducc combines the benefits of row-based and apriori-wise column-based techniques (see Section 4.4 for details), which allows it to perform by orders of magnitude faster than the above mentioned existing work.

There exist other techniques that are related to unique discovery. Grahne and Zhu present an apriori approach for discovering approximate keys within XML data [13]. Their algorithm evaluates discovered key candidates by the metrics support and confidence. As a side effect of using position lists to keep track of duplicate values in Ducc, one can easily extend Ducc to support the discovery of approximate keys. Moreover, the discovery of FDs [15,16,19] is also very similar to the problem of discovering uniques, as uniques functionally determine all other individual columns within a table. Thus, some approaches for unique discovery incorporate the knowledge on existing FDs [1,24]. Saiedian and Spencer presented an FD-based technique that supports unique discovery by identifying columns that are definitely part of all uniques and columns that are never part of any unique [24]. They showed that given a minimal set of FDs, any column that appears only on the left side of a FD must be part of all keys. In contrast, any column that appears only on the right side of a FD cannot be part of any key. It is worth noting that considering FDs for pruning the lattice more aggressively is complementary to Ducc. Also related to discovery of Ucs are the discovery of conditional functional dependencies (CFDs) [7, 9, 12], inclusion dependencies (INDs) [4, 21, 27] and conditional inclusion dependencies (CINDs) [3, 5]. However, like Gordian and HCA (but in contrast to Ducc), all these techniques depend on the number of attributes and hence they do not scale up to big datasets. Finally, more references and discussions on more general concepts of strict, approximate, and fuzzy dependencies can be found in [6, 8, 18]. But, all these works are orthogonal to the techniques presented in this paper.

9. CONCLUSION

In this paper, we addressed the problem of finding all unique and non-unique column combinations in big datasets (i.e., in datasets having several millions rows and hundreds attributes). The unique discovery problem is quite challenging given the exponential number of column combinations to check. In fact, it has been shown to be an NP-hard problem [14]. Existing approaches propose efficient techniques, but none are designed with very large datasets in mind [1, 10, 14, 16, 25]. Therefore, all these techniques strongly depend on the number of columns and hence they suffer from scalability issues.

We presented Ducc, a highly scalable and efficient approach to find (non-)unique column combinations in very large datasets. Ducc uses a novel hybrid graph traversal technique, which is a combination of the depth-first and random walk strategies. Ducc starts traversing the graph in a depth-first manner until it finds the first unique and then it starts following the boundary between uniques and non-uniques in a random walk manner. This allows Ducc to mainly depend on the solution set size rather than on the number of columns. Ducc also uses several light-weight structures that allow it to perform uniqueness checks in just a few milliseconds. As a result of its graph traversal technique and its light-weight data structures, Ducc can efficiently find uniques and non-uniques in very large datasets. Additionally, Ducc runs on several CPU cores and computing nodes to achieve even a higher scalability.

We evaluated Ducc using two real-world and one synthetic datasets and compared it with two state-of-the-art systems: Gordian and HCA. The results show the high superiority of Ducc over both; Ducc is up to 631x faster than Gordian and up to 398x faster than HCA. A series of scalability experiments showed the efficiency of Ducc to scale up to several CPU cores and to scale out to multiple computing nodes. Also, our experimental results showed that Ducc mainly depends on the solution set size and not on the number of columns. As a result, Ducc can achieve what before was not possible: processing datasets with hundreds of columns and many millions of records.

10. REFERENCES

[1] Z. Abedjan and F. Naumann. Advancing the discovery of unique column combinations. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 1565–1570, 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, 1994.

[3] J. Bauckmann, Z. Abedjan, U. Leser, H. Müller, and F. Naumann. Discovering conditional inclusion dependencies. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 2094–2098, 2012.

[4] J. Bauckmann, U. Leser, F. Naumann, and V. Tietz. Efficiently detecting inclusion dependencies. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1448–1450, 2007.

[5] L. Bravo, W. Fan, and S. Ma. Extending dependencies with conditions. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 243–254, 2007.

[6] P. G. Brown and P. J. Haas. BHUNT: Automatic discovery of fuzzy algebraic constraints in relational data. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 668–679, 2003.

[7] G. Cormode, L. Golab, K. Flip, A. McGregor, D. Srivastava, and X. Zhang. Estimating the confidence of conditional functional dependencies. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 469–482, 2009.

[8] W. Fan. Dependencies revisited for improving data quality. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 159–170, 2008.

[9] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 23(5):683–698, 2011.

[10] 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. Technical report. Last accessed on 2013-02-21.

[11] C. Goble and R. Stevens. State of the nation in data integration for bioinformatics. *J. of Biomedical Informatics*, 41(5):687–693, 2008.

[12] L. Golab, H. Karloff, F. Korn, D. Srivastava, and B. Yu. On generating near-optimal tableaux for conditional functional dependencies. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1):376–390, 2008.

[13] 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, 2002.

[14] 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.

[15] 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, 1998.

[16] 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.

[17] IBM InfoSphere Information Analyzer. `http://www-01.ibm.com/software/data/infosphere/information-analyzer`.

[18] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. CORDS: automatic discovery of correlations and soft functional dependencies. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 647–658, 2004.

[19] 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.

[20] Z. Lacroix and T. Critchlow. *Bioinformatics: managing scientific data*. Morgan Kaufmann, Burlington, MA, 2003.

[21] F. D. Marchi, S. Lopes, and J.-M. Petit. Unary and n-ary inclusion dependency discovery in relational databases. *Journal of Intelligent Information Systems*, 32(1):53–73, 2009.

[22] Microsoft Data Profiling Task, 03/2013. `http://msdn.microsoft.com/en-us/library/bb895263.aspx`.

[23] Oracle 11g Data Profiling. `http://www.oracle.com/technetwork/middleware/data-integration/index-082810.html`.

[24] 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.

[25] 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.

[26] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1150–1160, 2007.

[27] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava. On multi-column foreign key discovery. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1-2):805–814, 2010.