Einreichung für: <Konferenz/Workshop>,
Geplant als Veröffentlichung innerhalb der Lecture Notes in Informatics (LNI)

A Hybrid Approach for Efficient Unique Column Combination Discovery

Thorsten Papenbrock[1] Felix Naumann[2]

Abstract: Unique column combinations (UCCs) are groups of attributes in relational datasets that contain no value-entry more than once. Hence, they indicate keys and serve data management tasks, such as schema normalization, data integration, and data cleansing. Because the unique column combinations of a particular dataset are usually unknown, UCC discovery algorithms have been proposed to find them. All previous such discovery algorithms are, however, inapplicable to datasets of typical real-world size, e.g., datasets with more than 50 attributes and a million records.

We present the hybrid discovery algorithm HyUCC, which uses the same discovery techniques as the recently proposed functional dependency discovery algorithm HyFD: A hybrid combination of fast approximation techniques and efficient validation techniques. With it, the algorithm discovers *all minimal unique column combinations* in a given dataset. HyUCC does not only outperform all existing approaches, it also scales to much larger datasets.

Keywords: unique column combinations, data profiling, metadata, hybrid.

1 Unique Column Combinations

A unique column combination (UCC) is a set of attributes whose projection contains no duplicate entry. Knowing these unique combinations is particularly important when choosing key constraints for a given relational dataset, because the values in such columns uniquely identify all records in the dataset. Unique column combinations are moreover known to be useful for schema normalization, data cleansing, query optimization, schema reverse engineering, and many other tasks.

If not explicitly declared as key constraints, the UCCs of a particular dataset are typically unknown and need to be discovered. This is particularly true in "data lake" scenarios, which involve many external data sources. Data profiling is the computer science discipline that describes the investigation of a dataset for its metadata [AGN15]. The discovery of UCCs is an important profiling task, which has led to the development of various discovery algorithms [He13, Si06, AN11]. The task for these algorithms is to find *all minimal UCCs* that hold in a given relational instance. The search is restricted to minimal UCCs, i.e., sets of attributes from which no attribute can be removed without invalidating the uniqueness of the described column combination, because all non-minimal UCCs can easily be derived from the set of minimal UCCs. Furthermore, most use cases, such as database key discovery [Ma16], are interested in only the minimal UCCs.

[1] Hasso Plattner Institute (HPI), Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, thorsten.papenbrock@hpi.de
[2] Hasso Plattner Institute (HPI), Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, felix.naumann@hpi.de

The discovery of UCCs is a computational expensive task, because the search is NP-hard [Gu03] and even the solution space is exponential [He13]. For this reason, all known algorithms are limited to small datasets. We propose a new UCC discovery algorithm called HyUCC that takes UCC profiling a step forward, now being able to efficiently process datasets that are much larger than current limits. In fact, with HyUCC the discovery time becomes less of an issue than the ability of the executing machine to cope with the size of the UCC result set, which can grow exponentially large.

In a recent publication [PN16], we proposed the HyFD algorithm for the discovery of functional dependencies. Its fundamental ideas are also the basis for our new algorithm HyUCC, because FD discovery and UCC discovery are quite similar, i.e., many FD discovery techniques can also be used to profile UCCs and vice versa. So the main contributions of this short paper can be summarized as follows:

1. *Hybrid UCC algorithm*. We present a hybrid algorithm for the discovery of all minimal unique column combinations in relational datasets: The algorithm combines known row- and column-efficient techniques to cope with both long and wide datasets.

2. *Evaluation*. We evaluate our algorithm on several datasets demonstrating its superiority over existing UCC discovery algorithms. The experiments show that the algorithm is capable of computing both large numbers of rows and columns.

We first discuss related work in Section 2. Then, Section 3 introduces the intuition of our hybrid approach. Section 4 describes how these ideas can be implemented by explaining the differences to HyFD. In Section 5, we evaluate our algorithm and conclude in Section 6.

2 Related Work

There are basically two classes of UCC discovery algorithms: row-based discovery algorithms, such as GORDIAN [Si06], and column-based algorithms, such as HCA [AN11]. Row-based algorithms compare pairs of records in the dataset, derive so-called agree or disagree sets, and finally derive the UCCs from these sets. This discovery strategy performs well with increasing numbers of attributes, but falls short when the number of rows is high. Column-based algorithms model the search space as a powerset lattice and then traverse this lattice to identify the UCCs. The traversal strategies usually differ, but all algorithms of this kind make extensive use of pruning rules, i.e., they remove subsets of falsified candidates from the search space (these must be false as well) and supersets of validated candidates (which must be valid and not minimal). The column-based family of discovery algorithms scales well with larger numbers of records, but large numbers of attributes render them infeasible. Because both row- and column-efficient algorithms have their strengths, we combine these two search strategies in our HyUCC algorithm.

The currently most efficient UCC discovery algorithm is DUCC [He13], a column-based algorithm. The algorithm finds UCCs with a random walk approach through the search space lattice making maximum use of the known pruning rules. Because this algorithm has shown to be faster than both GORDIAN and HCA, it serves as our evaluation baseline.

The HyFD algorithm is a hybrid FD discovery algorithm that already proposed to mix row- and column efficient discovery techniques in order to scale with both dimensions of a relational dataset [PN16]. In a sense, the HyUCC algorithm is the sister algorithm of HyFD, which we modified in certain selected components to find UCCs instead of FDs. The changes we made are presented in this paper; the achieved performance improvements are comparable, as our evaluation shows.

3 Hybrid UCC discovery

The core idea of hybrid UCC discovery is to combine techniques from column-based and row-based discovery algorithms into one algorithm that automatically switches back and forth between these techniques, depending on which technique currently performs better. The challenge for these switches is to decide when to switch and to convert the intermediate results from one model into the other model, which is necessary to let the strategies support each other. In the following, we first describe the two individual discovery strategies; then, we discuss when and how the intermediate results can be synchronized.

Row-efficient strategy. Column-based UCC discovery algorithms, which are the family of algorithms that perform well on many rows, model the search space as a powerset lattice of attribute combinations where each edge represents a UCC candidate. The search strategy is then a classification problem of labelling each node as non-UCC, minimal UCC, or non-minimal UCC. Figure 1 depicts an example lattice for five attributes A, B, C, D, and E with labeled nodes.

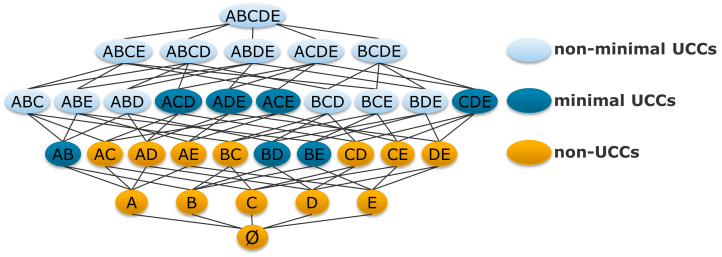

Fig. 1: UCC discovery in a powerset lattice.

For our hybrid algorithm, we propose a simple bottom-up traversal strategy: First, we test all candidates of size one, then of size two and so on. The lattice is generated level-wise using the *apriori-gen* algorithm [AS94]. Minimality pruning assures that no implicitly valid UCCs, i.e., non-minimal UCCs are ever generated [He13]. All discovered minimal UCCs must be stored as the algorithm's result.

An important characteristic of this discovery strategy is that all intermediate results during the discovery are correct but incomplete, that is, each discovered UCCs must be valid but not all UCCs have been discovered. Because correctness is guaranteed, we will always end the hybrid algorithm in a phase with this discovery strategy. Another characteristic of the bottom-up lattice traversal is that it might have to wade through many non-UCCs until

it reaches the true UCCs, because these are all placed along the virtual border between non-UCCs (below in the lattice) and true UCCs (above in the lattice). The fact that the number of these non-UCCs increases exponentially with the number of columns hinders algorithms of this family to scale well with increasing numbers of attributes – the lattice becomes extremely "wide". Hence, we need to utilize an alternative discovery strategy to skip most of the non-UCC nodes to reach the true UCCs faster.

Column-efficient strategy. Row-based / column-efficient UCC discovery strategies compare all records pair-wise and derive agree set from these comparisons. An agree set is a negative observation, i.e., a set of attributes that have same values in the two compared records and can, hence, not be a UCC; so agree sets correspond to non-UCCs in the attribute lattice. When all (or some) agree sets have been collected, there are efficient techniques to turn them into true UCCs [FS99].

A major weakness of this discovery strategy is that comparing all records pair-wise is usually infeasible. So suppose we stop the comparison of records at some time during the discovery; we basically compare only a sample r' of all r record pairs. When turning whatever agree sets we found so far into UCCs, these UCCs are most likely not all correct, because sampling might have missed some important agree sets. However, the intermediate result has three important properties (see [PN16] for proofs):

1. *Completeness:* Because all supersets of UCCs in the result are also assumed to be correct UCCs, the set of r'-UCCs is complete: It implies the entire set of r-UCCs, i.e., we find at least one X' in the r'-UCCs for each valid X in r-UCCs with $X' \subseteq X$.
2. *Minimality:* If a minimal r'-UCC is truly valid, then the UCC must also be minimal with respect to the real result. For this reason, the early stopping cannot lead to non-minimal or incomplete results.
3. *Proximity:* If a minimal r'-UCC is in fact invalid for the entire r, then the r'-UCC is still *close* to one or more valid specializations. In other words, most r'-UCCs need fewer specializations to reach the true r-UCCs on the virtual border than the unary UCCs at the bottom of the lattice so that the stopping approximates the real UCCs.

Hybrid strategy. For the hybrid UCC discovery strategy, we refer to the column-efficient search as the *sampling phase*, because we inspect carefully chosen subsets of record pairs for agree sets, and to the row-efficient search as the *validation phase*, because this phase directly validates individual UCC candidates. Intuitively, the hybrid discovery uses the sampling phase to jump over possibly many non-UCCs and the validation phase to produce a valid result. We obviously start with the sampling phase, switch back and forth between phases, and finally end with the validation phase. The questions that remain are *when* and *how* to switch between the phases.

The best moment to leave the sampling phase is when most of the non-UCCs have been identified and finding more non-UCCs becomes more expensive than simply directly checking their candidates. Of course, this moment is known neither a-priori nor during the process, because one would need to know the result already to calculate the moment. For

this reason, we switch optimistically back and forth whenever a phase becomes *inefficient*: The sampling becomes inefficient, when the number of newly discovered agree sets per comparison falls below a certain threshold; the validation becomes inefficient, when the number of valid UCCs per non-UCC falls below a certain threshold. With every switch, we relax this threshold a bit, so that the phases are considered efficient again. In this way, the hybrid discovery always progressively pursues the currently most efficient strategy.

To exchange intermediate results between phases, the hybrid algorithm must maintain all currently valid UCCs in some central data structure (we later propose a prefix-tree). When switching from sampling to validation, we update this central data structure of UCCs with the discovered agree sets. This means that we replace every single UCC for which a negative observation, i.e., an agree set exists with its valid, minimal refinements. The validation phase, then, directly operates on this data structure so that many non-UCCs are already excluded from the validation procedure. When switching from the validation to the sampling, the algorithm must not explicitly update the central data structure, because the validation already performs all changes directly to it. However, the validation automatically identifies record pairs that violated certain UCC candidates, and these record pairs should be suggested to the sampling phase for full comparisons as it is very likely that they indicate larger agree sets. In this way, both phases benefit from one another.

4 The HyUCC algorithm

We now describe our implementation of the hybrid UCC discovery strategy HyUCC. Because this algorithm is similar to the hybrid discovery algorithm HyFD, we omit certain details that can be found in [PN16]. The differences that make HyUCC discover unique column combinations instead of functional dependencies are in particular a prefix tree (trie) to store the UCCs, a UCC-specific validation, and UCC-specific pruning rules.

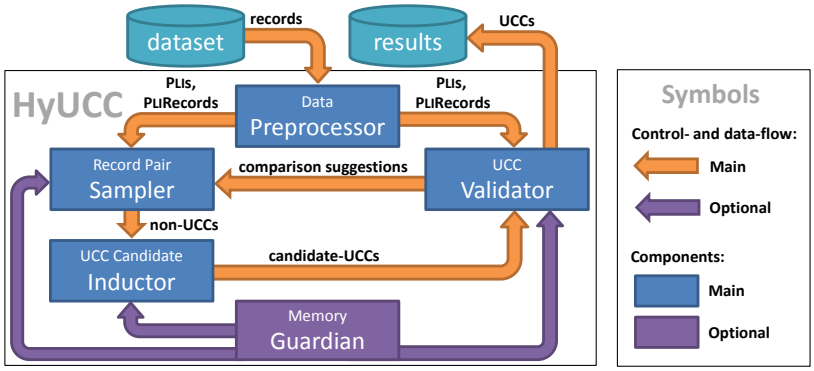

Fig. 2: Overview of HyUCC and its components.

Figure 2 gives an overview of HyUCC and its components. Given a relational dataset as a collection of records, the algorithm first runs them through a `Preprocessor` component that transforms the records into two smaller index structures: PLIs and PLIRecords. Then, HyUCC starts the sampling phase in the `Sampler` component. When the sampling has

become inefficient, the algorithms passes the discovered agree sets, i.e., the non-UCCs, to the `Inductor` component, which turns them into candidate-UCCs: UCCs that hold true on the sample of record pairs that was seen so far. Afterwards, the algorithm switches to the validation phase in the `Validator` component. This component systematically checks and creates candidate-UCCs. If the checking becomes inefficient, HyUCC switches back into the sampling phase handing over a set of comparison suggestions; otherwise, the validation continues until all candidates have been checked and all true UCCs can be returned. We now discuss the components of HyUCC in more detail.

Data Preprocessor. To determine a unique column combination, HyUCC must know the positions of same values in each attribute; it must not know the values itself. For this reason, the `Preprocessor` component transforms all records into the well-known and compact *position list indexes* (PLIs) data structure (also known as stripped partitions [Hu99]). A PLI is a set of record ID sets, where each record in a set has the same value in the attribute described by the PLI. Using the PLIs, the `Preprocessor` also creates PLIRecords, which are the records from the input dataset with dictionary compressed values. These are needed for the record comparisons in the `Sampler` component.

Record Pair Sampler. The `Sampler` component compares the PLIRecords to derive agree sets, i.e., non-UCCs. As stated earlier, a non-UCC is simply a set of attributes that have same values in two records. Because the sampling phase should be maximally efficient, the `Sampler` chooses the record pairs for the comparisons deliberately: Record pairs that are more likely to reveal non-UCCs are progressively chosen earlier in the process and less promising record pairs later. Intuitively, the more values two records share, the higher their probability of delivering a new non-UCC is. Vice versa, records that do not share any values cannot deliver any non-UCC and should not be compared at all.

So because the PLIs already group records with at least one identical value, HyUCC only compares records within same PLI clusters. For all records within same PLI clusters, we must however define a possibly efficient comparison order. For this purpose, the `Sampler` component first sorts all clusters in all PLIs with a different sorting key (see [PN16] for details). This produces different neighborhoods for each record in each of the record's clusters, even if the record co-occurs with same other records in its clusters. After sorting, the `Sampler` iterates all PLIs and compares each record to its direct neighbor. In this step, the algorithm also calculates the *number of discovered non-UCCs per comparison* for each PLI. This number indicates the sampling efficiency achieved with this particular PLI. In a third step, the `Sampler` can then rank the different PLIs by their efficiency, pick the most efficient PLI and use it to compare each record to its second neighbor. This comparison run updates the efficiency of the used PLI, so that it is re-ranked with the other. The `Sampler` then again picks the most efficient PLI for the next round of comparisons. This process of comparing records to their $n+1$ next neighbors progressively chooses most promising comparisons; it continues until the top ranked PLI is not efficient any more, which is the condition to switch to the validation phase.

UCC Candidate Inductor. The `Inductor` component updates the intermediate result of UCCs with the non-UCCs from the `Sampler` component. We store these UCCs in a prefix

tree, i.e., trie, where each node represents exactly one attribute and each path a UCC. Such a *UCC tree* allows for fast subset-lookups, which is the most frequent operation on the intermediate results of UCCs. The UCC tree in HyUCC is much leaner than the FD tree used in HyFD, because no additional right-hand-sides must be stored in the nodes; the paths alone suffice to identify the UCCs.

Initially, the UCC tree contains all individual attributes, assuming that each of them is unique. The `Inductor` then refines this initial UCC tree with every non-UCC that it receives from the `Sampler`: For every non-UCC, remove the UCC and all of its subsets from the UCC tree, because these must all be non-unique. Then, create all possible specializations of each removed non-UCC by adding one additional attribute; these could still be true UCCs. For each specialization, check the UCC tree for existing subsets (generalizations) or supersets (specialization). If a generalization exists, the created UCC is not minimal; if a specialization exists, it is invalid. In both cases, we ignore the generated UCC; otherwise, we add it to the UCC tree as a new candidate.

UCC Validator. The `Validator` component traverses the UCC tree level-wise from bottom (individual attributes) to top (union of all attributes). This traversal is implemented as a simple breadth-first search. Each leaf-node represents a UCC candidate X that the algorithm validates. If the validation returns a positive result, the `Validator` keeps the UCC in the lattice; otherwise, it removes the non-UCC X and adds all XA to the tree with $A \notin X$ and XA is both minimal (XA has no specialization in the UCC tree) and valid (XA has no generalization in the UCC tree). After validating a level of UCC candidates, the `Validator` calculates the number of valid UCCs per validation. If this efficiency value does not meet a current threshold, HyUCC switches back into the sampling phase; the `Validator`, then, continues with the next level when it gets the control flow back.

To validate a column combination X, the `Validator` intersects the PLIs of all columns in X. Intersecting a PLI with one or more other PLIs means to intersect all the record clusters that they contain [Hu99]. The result is again a PLI. If this PLI contains no clusters of size greater than one, its column combination X is unique; otherwise, X is non-unique and the records in the clusters greater than one violate it. The algorithm suggests these records to the `Sampler` as interesting comparison candidates, because they have not yet been compared and may reveal additional non-UCCs of greater size.

An efficient way to calculate the PLI intersections is the following: First, pre-calculate the inverse of each PLI (this is done in the `Preprocessor` already). Then, take the PLI with the fewest records as a *pivot* PLI (recall that PLIs do not contain clusters of size one so that the numbers of records usually differ). This pivot PLI requires the least number of intersection look-ups to become unique. For each cluster in the pivot PLI do the following: Iterate all record IDs and, for each record ID, look-up the cluster numbers in the inverted PLIs of all other attributes in X; store each retrieved list of cluster numbers in a set. If this set already contains a cluster number sequence equal to the sequence the `Validator` wants to insert, then the algorithm found a violation and can stop the validation process for this candidate. In this case, the current record and the record referring to the cluster number sequence in the set are sent to the `Sampler` as a new comparison suggestion.

Memory Guardian. The `Guardian` is an optional component that monitors the memory consumption of HyUCC. If at any point the memory threatens to become exhausted, this component gradually reduces the maximum size of UCCs in the result until sufficient memory becomes available. Of course, the result is then not complete any more, but correctness and minimality of all reported UCCs is still guaranteed. Also, the result limitation only happens if the *result* becomes so large that the executing machine cannot store it any more. Other algorithms would simply break in such cases. To reduce the size, the `Guardian` deletes all agree sets and UCCs that exceed a certain maximum size. It then forbids further insertions of any new elements of this or greater size.

5 Evaluation

We evaluate and compare HyUCC to its sister algorithm HyFD [PN16] and to the UCC discovery algorithm DUCC, which the authors have shown to be superior over other approaches [He13]. All three algorithms have been implemented for the *Metanome* data profiling framework, which defines standard interfaces for profiling algorithms [Pa15]. The algorithms and the framework are available online[3]. Our experiments use a Dell PowerEdge R620 with two Intel Xeon E5-2650 2.00 GHz CPUs and 128 GB RAM. The server runs on CentOS 6.4 and uses OpenJDK 64-Bit Server VM 1.7.0_25 as Java environment. Details about our experimental datasets can be found in [PN16] and on our repeatability website[4]. Note that the experiments use the `null` $=$ `null` semantics, because this was also used in related work; HyUCC can compute UCCs with `null` $\neq$ `null` as well, which makes the search faster, because columns with `null` values become unique more quickly.

5.1 Varying the datasets

In this experiment, we measure the discovery times for the three algorithms on eight real-world datasets. The datasets, their characteristics, and the runtimes are listed in Table 1. The results show that HyUCC usually outperforms the current state-of-the-art algorithm DUCC by orders of magnitude: On most datasets, HyUCC is about 4 times faster than DUCC; on the *flight* dataset, it is even more than 1,000 times faster. Only on *zbc00dt* DUCC is slightly faster, because only one UCC was to be discovered and DUCC does not pre-compute PLIRecords or inverted PLIs as HyUCC does. Furthermore, the runtimes of HyFD show that UCC discovery is considerably faster than FD discovery. Because HyFD and HyUCC use similar discovery techniques, this speedup is due to the smaller result sets.

Because we can easily parallelize the validations in HyUCC and HyFD, the experiment also lists the runtimes for their parallel versions. With 32 cores available, the parallel algorithms use the same number of threads. On the given datasets, HyUCC could in this way achieve 1.2 (*uniprot*) to more than 9 (*isolet*) times faster runtimes than its single-threaded version (less than 32, because only the validations run in parallel).

[3] www.metanome.de
[4] www.hpi.de/naumann/projects/repeatability.html

Dataset	Cols [#]	Rows [#]	Size [MB]	FDs [#]	UCCs [#]	DUCC	HyFD	HyUCC	HyFD parallel	HyUCC parallel
ncvoter	19	8 m	1,263.5	822	96	706.1	1,009.6	220.1	239.8	157.9
hepatitis	20	155	0.1	8,250	348	0.6	0.4	0.1	0.4	0.1
horse	27	368	0.1	128,727	253	0.8	5.8	0.2	3.7	0.2
zbc00dt	35	3 m	783.0	211	1	57.7	191.1	58.2	69.4	58.2
ncvoter_c	71	100 k	55.7	208,470	1,980	170.3	2,561.6	51.3	533.4	14.9
ncvoter_s	71	7 m	4,167.6	>5 m	32,385	>8 h	>8 h	>8 h	>8 h	5,870.2
flight	109	1 k	0.6	982,631	26,652	4,212.5	54.1	3.7	19.5	1.5
uniprot	120	1 k	2.1	>10 m	1,973,734	>8 h	>1 h	92.5	>1 h	76.7
isolet	200	6 k	12.9	244 m	1,241,149	>8 h	1,653.7	410.9	482.3	45.1

Tab. 1: Runtimes in seconds for several real-world datasets

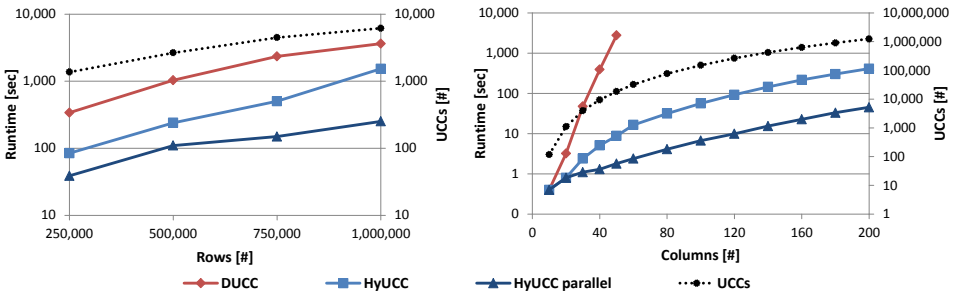

Fig. 3: Row scalability on *ncvoter_s* (71 columns) and column scalability on *isolet* (6238 rows).

5.2 Varying columns and rows

We now evaluate the scalability of HyUCC with the input's number of rows and columns. The row-scalability is evaluated on the *ncvoter_s* dataset with 71 columns and the column-scalability on the *isolet* dataset with 6238 rows. Figure 3 shows the measurements for DUCC and HyUCC. The measurements also include the runtimes for the parallel version of HyUCC; the dotted line indicates the number of UCCs for each measurement point.

The graphs show that the runtimes of both algorithms scale well with the number of UCCs in the result, which is a desirable discovery behavior. However, HyUCC still outperforms DUCC in both dimensions – even in the row-dimension that DUCC is optimized for: It is about 4 times faster in the row-scalability experiment and 4 to more than 310 times faster in the five column-scalability measurements that we could create for DUCC. HyUCC's advantage in the column-dimension is clearly the fact that the non-UCCs derived from the sampling phase allow the algorithm to skip most of the lower-level UCC candidates (and the number of these candidates increases exponentially with the number of columns); the advantage in the row-dimension is also this sampling phase of HyUCC, allowing it to skip many candidates and, because the number of UCCs also increases when increasing the number of rows, this gives HyUCC a significant advantage.

6 Conclusion & Future Work

In this paper, we proposed HyUCC, a hybrid UCC discovery algorithm that combines row- and column-efficient techniques to process relational datasets with both many records and many attributes. On most real-world datasets, HyUCC outperforms all existing UCC discovery algorithms by orders of magnitude.

For future work, we suggest to find novel techniques to deal with the often huge amount of results. Currently, HyUCC limits its results if these exceed main memory capacity, but one might consider using disk or flash memory in addition for these cases.

References

[AGN15] Abedjan, Ziawasch; Golab, Lukasz; Naumann, Felix: Profiling relational data: a survey. VLDB Journal, 24(4):557–581, 2015.

[AN11] Abedjan, Ziawasch; Naumann, Felix: Advancing the Discovery of Unique Column Combinations. In: Proceedings of the International Conference on Information and Knowledge Management (CIKM). pp. 1565–1570, 2011.

[AS94] Agrawal, Rakesh; Srikant, Ramakrishnan: Fast Algorithms for Mining Association Rules in Large Databases. In: Proceedings of the International Conference on Very Large Databases (VLDB). pp. 487–499, 1994.

[FS99] Flach, Peter A; Savnik, Iztok: Database dependency discovery: a machine learning approach. AI Communications, 12(3):139–160, 1999.

[Gu03] Gunopulos, Dimitrios; Khardon, Roni; Mannila, Heikki; Saluja, Sanjeev; Toivonen, Hannu; Sharma, Ram Sewak: Discovering All Most Specific Sentences. ACM Transactions on Database Systems (TODS), pp. 140–174, 2003.

[He13] Heise, Arvid; Quiane-Ruiz, Jorge-Arnulfo; Abedjan, Ziawasch; Jentzsch, Anja; Naumann, Felix: Scalable Discovery of Unique Column Combinations. Proceedings of the VLDB Endowment, 7(4):301–312, 2013.

[Hu99] Huhtala, Ykä; Kärkkäinen, Juha; Porkka, Pasi; Toivonen, Hannu: TANE: An efficient algorithm for discovering functional and approximate dependencies. The Computer Journal, 42(2):100–111, 1999.

[Ma16] Mancas, Christian: Perspectives in Business Informatics Research. Springer, chapter Algorithms for Database Keys Discovery Assistance, pp. 322–338, 2016.

[Pa15] Papenbrock, Thorsten; Bergmann, Tanja; Finke, Moritz; Zwiener, Jakob; Naumann, Felix: Data Profiling with Metanome. Proceedings of the VLDB Endowment, 8(12):1860–1871, 2015.

[PN16] Papenbrock, Thorsten; Naumann, Felix: A Hybrid Approach to Functional Dependency Discovery. In: Proceedings of the International Conference on Management of Data (SIGMOD). 2016.

[Si06] Sismanis, Yannis; Brown, Paul; Haas, Peter J.; Reinwald, Berthold: GORDIAN: Efficient and Scalable Discovery of Composite Keys. In: Proceedings of the VLDB Endowment. pp. 691–702, 2006.