Hitting Set Enumeration with Partial Information for Unique Column Combination Discovery

Johann Birnick[⋆], Thomas Bläsius[♦], Tobias Friedrich[♦],
Felix Naumann[♦], Thorsten Papenbrock[♦], Martin Schirneck[♦]

[⋆]ETH Zurich, Zurich, Switzerland

[♦]Hasso Plattner Institute, University of Potsdam, Germany
firstname.lastname@hpi.de

ABSTRACT

Unique column combinations (UCCs) are a fundamental concept in relational databases. They identify entities in the data and support various data management activities. Still, UCCs are usually not explicitly defined and need to be discovered. State-of-the-art data profiling algorithms are able to efficiently discover UCCs in moderately sized datasets, but they tend to fail on large and, in particular, on wide datasets due to run time and memory limitations.

In this paper, we introduce `HPIValid`, a novel UCC discovery algorithm that implements a faster and more resource-saving search strategy. `HPIValid` models the metadata discovery as a hitting set enumeration problem in hypergraphs. In this way, it combines efficient discovery techniques from data profiling research with the most recent theoretical insights into enumeration algorithms. Our evaluation shows that `HPIValid` is not only orders of magnitude faster than related work, it also has a much smaller memory footprint.

PVLDB Reference Format:
Johann Birnick, Thomas Bläsius, Tobias Friedrich, Felix Naumann, Thorsten Papenbrock, Martin Schirneck. Hitting Set Enumeration with Partial Information for Unique Column Combination Discovery. *PVLDB*, 13(11): 2270-2283, 2020.
DOI: https://doi.org/10.14778/3407790.3407824

1. INTRODUCTION

Keys are among the most fundamental type of constraint in relational database theory. A *key* is a set of attributes whose values uniquely identify every record in a given relational instance. This uniqueness property is necessary to determine entities in the data. Keys serve to query, link, and merge entities. A *unique column combination* (UCC) is the observation that in a given relational instance a certain set of attributes S does not contain any duplicate entries. In other words, the multiset projection of schema R on S contains only unique entries. Because UCCs describe attribute sets that fulfill the necessary properties of a key, whereby `null` values require special consideration, they serve to define key constraints and are, hence, also called *key candidates*.

In practice, key discovery is a recurring activity as keys are often missing for various reasons: on freshly recorded data, keys may have never been defined, on archived and transmitted data, keys sometimes are lost due to format restrictions, on evolving data, keys may be outdated if constraint enforcement is lacking, and on fused and integrated data, keys often invalidate in the presence of key collisions. Because UCCs are also important for many data management tasks other than key discovery, such as anomaly detection, data integration, data modeling, query optimization, and indexing, the ability to discover them efficiently is crucial.

Data profiling algorithms automatically discover metadata, like unique column combinations, from raw data and provide this metadata to any downstream data management task. Research in this area has led to various UCC discovery algorithms, such as `GORDIAN` [34], `HCA` [2], `DUCC` [22], `Swan` [3], and `HyUCC` [33]. Out of these algorithms, only `HyUCC` can process datasets of multiple gigabytes in size in reasonable time, because it combines the strengths of all previous algorithms (lattice search, candidate inference, and parallelization). At some point, however, even `HyUCC` fails to process certain datasets, because the approach needs to maintain an exponentially growing search space in main memory, which eventually either exhausts the available memory or, due to search space maintenance, the execution time.

There is a close connection between the UCC discovery problem and the *enumeration of hitting sets in hypergraphs*. Hitting set-based techniques were among the first tried for UCC discovery [29] and still play a role today in the form of row-based algorithms. To utilize the hitting set connection directly, one needs to feed the algorithm with information on all pairs of records of the database [1, 35]. This is prohibitive for larger datasets due to the quadratic scaling.

In this paper, we propose a novel approach that automatically discovers all unique column combinations in any given relational dataset. The algorithm is based on the connection to hitting set enumeration, but additionally uses the key insight that most of the information in the record pairs is redundant and that the following strategy of utilizing the state-of-the-art hitting set enumeration algorithm `MMCS` [31] finds the relevant information much more efficiently. We start the discovery with only partial (or even no) information about the database. Due to this lack of information, the

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407824

resulting solution candidates might not actually be UCCs. A validation then checks whether the hitting set is a true UCC. If not, the validation additionally provides a reason for its incorrectness, pointing directly to the part of the database that MMCS needs to avoid the mistake. Moreover, we show that all previous decisions of MMCS would have been exactly the same, even if MMCS had full information about all row pairs of the database. Thus, our new approach can include the new information on the fly and simply resume the enumeration where it was before. Instead of providing full and probably redundant information about the database to the enumeration subroutine, the enumeration decides for itself which information is necessary to make the right decisions. Due to its hitting set-based nature, we named our algorithm ***H**itting set enumeration with **P**artial **I**nformation and **Valid**ation*, HPIValid for short. Our approach adopts state-of-the-art techniques from both data profiling and hitting set enumeration bringing together the two research communities. We add the following contributions:

(1) **UCC discovery.** We introduce HPIValid, a novel UCC discovery algorithm that outperforms state-of-the-art algorithms in terms of run time and memory consumption.
(2) **Hitting set enumeration with partial information.** We prove that the hitting set enumeration algorithm MMCS remains correct when run on a partial hypergraph, provided that there is a validation procedure for candidate solutions. We believe that this insight can be key to also solve similar task like the discovery of functional dependencies or denial constraints.
(3) **Subset closedness.** We introduce the concept of subset closedness of hitting set enumeration algorithms, and prove that this property is sufficient for enumeration with partial information to succeed. This makes it easy to replace MMCS with a different enumeration procedure, as long as it is also subset-closed.
(4) **Sampling strategy.** We propose a robust strategy how to extract the relevant information from the database efficiently in the presence of redundancies.
(5) **Exhaustive evaluation.** We evaluate our algorithm on dozens of real-world datasets and compare it to the current state-of-the-art UCC discovery algorithm HyUCC.

Besides being able to solve instances that were previously out of reach for UCC discovery, HPIValid is up to three orders of magnitude faster than a non-parallel running HyUCC and has a smaller memory footprint.

The requirement of subset closedness makes our novel algorithm harder to parallelize, but offers a much more resource and environmentally friendly run time profile. We focus on the discovery of *minimal* UCCs, from which all UCCs can easily be inferred by simply adding arbitrary.

Next, we present related work in Section 2. Section 3 introduces the paradigm of hitting set enumeration with partial information and how to apply it to UCC discovery. Section 4 describes the components of HPIValid, in Section 5 we evaluate our algorithm. We conclude the paper in Section 6.

2. RELATED WORK

Due to the importance of keys in the relational data model, much research has been conducted on finding keys in a given relational instance. Early research on key discovery, such as [16], is in fact almost as old as the relational model itself. Beeri et al. have shown that deciding whether there exists a key of cardinality less than a given value in a given relational instance is an NP-complete problem [5]. Moreover, the discovery of a key of minimum size in the database is also likely not fixed-parameter tractable as it is $W[2]$-complete when parameterized by the size [18]. Finding all keys or unique column combinations is computationally even harder. For this reason, only few automatic data profiling algorithms exist that can discover all unique column combinations. The discovery problem is closely related to the enumeration of hitting sets (a.k.a. the transversal hypergraph problem, frequent itemset mining, or monotone dualization, see [15]). Many data profiling algorithms, including the first UCC discovery algorithms, rely on the hitting sets of certain hypergraphs; the process of deriving complete hypergraphs, however, is a bottleneck in the computation [30].

In modern data profiling, one usually distinguishes two types of profiling algorithms: row and column-based approaches. For a comprehensive overview and detailed discussions of the different approaches, we refer to [1] and [28]. Row-based algorithms, such as GORDIAN [34], advance the initial hitting set idea. They compare all records in the input dataset pair-wise to derive all valid, minimal UCCs. Column-based algorithms, such as HCA [2], DUCC [22], and Swan [3], in contrast, systematically enumerate and test individual UCC candidates while using intermediate results to prune the search space. The algorithms vary mostly in their traversal strategies, which are breadth-first bottom-up for HCA and a depth-first random-walk for DUCC. Both row and column-based approaches have their strengths: record comparisons scale well with the number of attributes, and systematic candidate tests scale well with the number of records. Hybrid algorithms aim to combine both aspects. The one proposed in [25] exploits the duality between minimal difference sets and minimal UCCs to mutually grow the available information about the search and the solution space. HyUCC [33] switches between column and row-based parts heuristically whenever the progress of the current approach is low. Hyb [35] is a hybrid algorithm for the discovery of embedded uniqueness constraints (eUCs), an extension of UCCs to incomplete data. It proposes special ideas tailored to incomplete data, but is based on the same discovery approach as HyUCC. All three algorithms share the caveat of a large memory footprint, HyUCC and Hyb need to store all discovered UCCs or eUCs during computation, the algorithm in [25] additionally tracks their minimal hitting sets.

Our algorithm HPIValid also implements a hybrid strategy, but instead of switching in level transitions, the column-based part of the algorithm enumerates preliminary solutions on partial information. The validation of those solution candidates then points to areas of the database where focused sampling can reveal new information. We show that this partial information approach succeeds to find all UCCs as if the row-based approach had been applied exhaustively. Similar to [25], the duality with the hitting sets problem is employed to find new relevant difference sets. However, the tree search with validation allows us to compute them without the need of holding all previous solutions in memory.

Regarding the transversal hypergraph problem, Demetrovics and Thi [14] as well as Mannila and Räihä [29], independently, were the first to pose the task of enumerating all minimal hitting sets of a hypergraph. Interestingly, both raised the issue in the context of databases as they employed hitting sets in the discovery of hidden dependencies

between attributes. It was shown much later that also every hitting set problem can be translated into the discovery of UCCs in certain dataset, making the two problems equivalent [18]. Hypergraphs stemming from real-world databases are known to be particularly suitable for enumeration [10]. The applications of minimal hitting sets have grown far beyond data profiling to domains such as data mining, bioinformatics and AI. We refer the reader to the surveys [15, 19] for an overview. `MMCS` is currently the fastest hitting set enumeration algorithm on real-world instances, see [19].

3. HITTING SET ENUMERATION WITH PARTIAL INFORMATION

There is an intimate connection between unique column combinations in relational databases and hitting sets in hypergraphs [1, 7, 13, 18, 25]. Intuitively, when comparing any two records of the database, a UCC must contain at least one attribute in which the records disagree, otherwise, they are indistinguishable. When viewing these difference sets of record pairs as edges of a hypergraph, its hitting sets correspond exactly to the UCCs of the database. This implies a two-step approach for UCC enumeration. First, compute the difference sets of all record pairs, and then apply one of the known algorithms for hitting set enumeration.

The enumeration step is generally a hard problem. However, the hypergraphs generated from real-world databases are usually well-behaved and allow for efficient enumeration [9, 10]. In fact, the bottleneck of the above approach is to compute all difference sets rather than the enumeration [10]. This observation contrasts sharply with the theoretical running time bounds, which are exponential for the hitting sets but polynomial for the difference sets [17].

The core idea of our algorithm is the following: We first sample a few record pairs and compute their difference sets. This gives us a *partial hypergraph*, which might be missing some edges. However, we pretend for now that we already have the correct hypergraph and start the hitting set enumeration. Due to the partial information, the candidate solutions we find are no longer guaranteed to be UCCs. Thus, whenever we find a hitting set, we use *validation* to check whether it is a UCC of the original database. If not, we know that the partial hypergraph is in fact incomplete. In this case, the validation procedure provides new row pairs whose difference sets yield yet unknown edges. In a very simple approach, one could then include the new information and restart the enumeration with the updated hypergraph. This is repeated until the validation certifies that every hitting set we find is indeed a UCC.

To prove that this approach is successful, one needs to show that we obtain the true hypergraph until the algorithm terminates. Intuitively, this comes from the fact that missing edges in the partial hypergraph make the instance less constraint, meaning that the lack of information might lead to some unnecessary hitting sets, which are rejected by the validation, but the true UCCs are already present.

In the remainder of this section, we actually prove the much stronger statement that we can even eliminate the need of restarts. The crucial concept in the proofs is minimality: a hitting set is minimal, if no proper subset intersects all edges; a UCC is minimal if it contains no strictly smaller valid UCC. First, we show the *correctness* of our algorithm. If the validation procedure asserts that a minimal hitting set of the partial hypergraph is a UCC, then it must also be a *minimal* UCC. Even though we cannot be sure that we have all relevant information yet, we can already start returning the found solution to the user. Secondly, we show *completeness*, meaning that we indeed found all minimal UCCs once the algorithm terminates. Thirdly, we define what it means for an enumeration algorithm to be *subset-closed*. We then show that subset-closed algorithms do not need to be restarted on a failed validation. Instead, they can simply resume the enumeration with an updated partial hypergraph. Finally, we note that the `MMCS` [31] algorithm we use to enumerate hitting sets is indeed subset-closed.

Before we prove these results, we introduce some notation and our null semantics in Section 3.1. A detailed description of `HPIValid` can be found in Section 4.

3.1 Notation and Null Semantics

Hypergraphs and hitting sets. A *hypergraph* is a finite *vertex set* V together with a system of subsets $\mathcal{H} \subseteq \mathcal{P}(V)$, the *(hyper-)edges*. We identify a hypergraph with its edgeset $\mathcal{H}$. $\mathcal{H}$ is a *Sperner hypergraph* if none of its edges is properly contained in another. The *minimization* of $\mathcal{H}$ is the subsystem of inclusion-wise minimal edges, $\min(\mathcal{H})$. For hypergraphs $\mathcal{H}$ and $\mathcal{G}$, we write $\mathcal{H} \prec \mathcal{G}$ if every edge of $\mathcal{H}$ contains an edge of $\mathcal{G}$. This is a preorder, i.e., $\prec$ is reflexive and transitive. On Sperner hypergraphs, $\prec$ is also antisymmetric. A *transversal* or *hitting set* for $\mathcal{H}$ is a subset $T \subseteq V$ of vertices such that T has a non-empty intersection with every edge $E \in \mathcal{H}$. A hitting set is *(inclusion-wise) minimal* if it does not properly contain any other hitting set. The minimal hitting sets for $\mathcal{H}$ form the *transversal hypergraph* $\mathrm{Tr}(\mathcal{H})$. $\mathrm{Tr}(\mathcal{H})$ is always Sperner. For the transversal hypergraph, it does not make a difference whether the full hypergraph is considered or its minimization, $\mathrm{Tr}(\min(\mathcal{H})) = \mathrm{Tr}(\mathcal{H})$. Also, it holds that $\mathrm{Tr}(\mathrm{Tr}(\mathcal{H})) = \min(\mathcal{H})$.

Relational databases. A *relational schema* R is a finite set of *attributes* or *columns*. A *record* or *row* is a tuple whose entries are indexed by R, a *(relational) database* $\mathfrak{r}$ *(over R)* is a finite set of such rows. For any row $r \in \mathfrak{r}$ and any subset $S \subseteq R$ of columns, $r[S]$ denotes the subtuple of r projected on S and $r[a]$ is the *value* of r at attribute a.

For any two rows $r_1, r_2 \in \mathfrak{r}$, $r_1 \neq r_2$, their *difference set* $\{a \in R \mid r_1[a] \neq r_2[a]\}$ is the set of attributes in which the rows have different values. We denote the hypergraph of minimal difference sets by $\mathcal{D}$. A *unique column combination* (UCC) is a subset $S \subseteq R$ of attributes such that for any two rows r_1, r_2, there is an attribute $a \in S$ such that $r_1[a] \neq r_2[a]$. A UCC is *(inclusion-wise) minimal* if it does not properly contain any other UCC. The (minimal) UCCs are exactly the (minimal) hitting sets of $\mathcal{D}$. Listing all minimal UCCs is the same as enumerating $\mathrm{Tr}(\mathcal{D})$.

Null semantics. Relational data may exhibit `null` values ($\perp$) to indicate *no value* [21]. This is an issue for the UCC validation procedure, because it needs to know how `null` values compare to other values including other `null` values. In this work, we follow the pessimistic `null` comparison semantics of related work [2, 3, 22, 33, 34], which defines that `null` $=$ `null` and `null` $\neq v$ for any non-`null` value v. More precise interpretations of `null` values for UCCs use *possible world* and *certain world* models [24], leading to specialized UCC definitions and discovery approaches [35]. In general, `null` semantics are an ongoing line of research [1, 4, 26], but not the focus of this study, hence the practical definition.

3.2 Correctness and Completeness

We need two things to show that the restart approach indeed enumerates the minimal UCCs of a database: first, an effective validation whether a minimal hitting set for the partial hypergraph is also a minimal UCC, and, secondly, the assertion that once the enumeration does not produce any more new edges, we have indeed found all UCCs. We do so by showing a general result about hypergraphs that yields the two assertions as corollaries. This lemma is of independent interest for hypergraph theory.

LEMMA 1. *For any two hypergraphs $\mathcal{H}$ and $\mathcal{G}$, we have $\mathcal{H} \prec \mathcal{G}$ if and only if $\mathrm{Tr}(\mathcal{H}) \succ \mathrm{Tr}(\mathcal{G})$.*

PROOF. First, let $T \in \mathrm{Tr}(\mathcal{G})$ be a minimal hitting set for $\mathcal{G}$. If every edge of $\mathcal{H}$ contains some edge of $\mathcal{G}$, T is also a hitting set for $\mathcal{H}$. T may not be minimal in that regard but it contains some minimal transversal from $\mathrm{Tr}(\mathcal{H})$. $\mathrm{Tr}(\mathcal{H}) \succ \mathrm{Tr}(\mathcal{G})$ follows from here.

For the other direction, the very same argument shows that $\mathrm{Tr}(\mathcal{H}) \succ \mathrm{Tr}(\mathcal{G})$ implies $\mathrm{Tr}(\mathrm{Tr}(\mathcal{H})) \prec \mathrm{Tr}(\mathrm{Tr}(\mathcal{G}))$, this is equivalent to $\min(\mathcal{H}) \prec \min(\mathcal{G})$. The proof is completed by applying the transitivity of the preorder $\prec$ and the two facts $\mathcal{H} \prec \min(\mathcal{H})$ and $\min(\mathcal{G}) \prec \mathcal{G}$. □

Recall that $\mathcal{D}$ denotes the collection of minimal difference sets of some database and that $\mathrm{Tr}(\mathcal{D})$ are the minimal UCCs. Let $\mathcal{P}$ be the current partial hypergraph consisting of the difference sets sampled so far. It may contain difference sets that are not globally minimal (or not even minimal in $\mathcal{P}$), i.e., $\mathcal{P} \not\subseteq \mathcal{D}$. Nevertheless, we always have $\mathcal{P} \prec \mathcal{D}$, because every difference set contains a minimal difference set. $\mathrm{Tr}(\mathcal{P})$ consists of the minimal hitting sets of the partial hypergraph. These are the candidates for validation.

COROLLARY 1. *A minimal hitting set for a partial hypergraph is a minimal UCC if and only if it is any UCC.*

PROOF. The implication from minimal UCC to any UCC is trivial. For the opposite direction, let $T \in \mathrm{Tr}(\mathcal{P})$ be a hitting set for the partial hypergraph such that T is a UCC of the database. T thus contains some *minimal* UCC $T' \in \mathrm{Tr}(\mathcal{D})$. $\mathcal{P} \prec \mathcal{D}$ holds and Lemma 1 shows that there exists some hitting set $E \in \mathrm{Tr}(\mathcal{P})$ such that $E \subseteq T'$. As $\mathrm{Tr}(\mathcal{P})$ is a Sperner hypergraph, we must have $T = E \subseteq T' \subseteq T$. All three sets are the same, T itself is the minimal UCC. □

COROLLARY 2. *If the enumeration of hitting sets for the partial hypergraph does not produce a new edge, we have indeed found all minimal UCCs.*

PROOF. If no call to the validation reveals a new unhit edge, we have reached the state $\mathrm{Tr}(\mathcal{P}) \subseteq \mathrm{Tr}(\mathcal{D})$. We need to show that this is sufficient for $\mathrm{Tr}(\mathcal{P}) = \mathrm{Tr}(\mathcal{D})$. The preorder $\prec$ generalizes set inclusion, thus $\mathrm{Tr}(\mathcal{P}) \subseteq \mathrm{Tr}(\mathcal{D})$ implies $\mathrm{Tr}(\mathcal{P}) \prec \mathrm{Tr}(\mathcal{D})$. Also, $\mathcal{P} \prec \mathcal{D}$ gives $\mathrm{Tr}(\mathcal{P}) \succ \mathrm{Tr}(\mathcal{D})$ via Lemma 1. Now $\mathrm{Tr}(\mathcal{P}) = \mathrm{Tr}(\mathcal{D})$ follows from the antisymmetry of $\prec$ on Sperner hypergraphs. □

3.3 Forgoing Restarts

Restarting the enumeration every time we discover a new unhit edge is obviously not ideal. The main obstacle to simply resuming the work with the updated hypergraph is the risk of overlooking solutions that are minimal UCCs of the database but not minimal hitting sets of the partial hypergraph. This is because past decisions may lock the algorithm out of regions in which the new update reveals undiscovered solutions. Next, we give a sufficient condition for overcoming these obstacles. Any algorithm that meets the condition can therefore be combined with our sampling scheme to solve the UCC problem without any restarts.

A hitting set enumeration method can be seen as a means to decide, at least implicitly, for vertex sets whether they are minimal hitting sets or not. We call an algorithm *subset-closed* if this decision is never made for a set before it is made for all of its subsets. Note that this does not mean that the algorithm needs to check every subset explicitly.

LEMMA 2. *Any subset-closed hitting set enumeration algorithm combined with a sampling scheme and validation discovers all minimal UCCs of a database without restarts.*

PROOF. We claim that a subset-closed algorithm does not overlook any minimal UCC, even if this solution is only revealed by some later update of the partial hypergraph $\mathcal{P}$. Recall that the transversal hypergraph $\mathrm{Tr}(\mathcal{D})$ contains exactly the minimal UCCs. To reach a contradiction, assume that the claim is false and let solution $T \in \mathrm{Tr}(\mathcal{D}) \setminus \mathrm{Tr}(\mathcal{P})$ be such that T corresponds to a node in the power set lattice that was discarded in a previous computation step. By construction, $\mathcal{P} \prec \mathcal{D}$ holds and Lemma 1 implies $\mathrm{Tr}(\mathcal{P}) \succ \mathrm{Tr}(\mathcal{D})$. In other words, there exists a hitting set $S \in \mathrm{Tr}(\mathcal{P})$ for the partial hypergraph such that $S \subsetneq T$. The algorithm is subset-closed, thus S was previously found and validated. This is a contradiction to T being a minimal UCC. □

There are many enumeration algorithms known for minimal hitting sets [19]. Currently the fastest algorithm in practice is the *Minimal-To-Maximal Conversion Search* (`MMCS`) by Murakami and Uno [31]. One can verify that `MMCS` is indeed a subset-closed algorithm, we can thus apply it to efficiently enumerate UCCs in the partial information setting.

4. ALGORITHM DESCRIPTION

We first give a high-level overview of our algorithm *Hitting Set Enumeration With Partial Information and Validation* (`HPIValid`) illustrated in Figure 1. The different components are subsequently explained in detail. The execution of `HPIValid` is split into four major phases. During *preprocessing*, the input table is read and brought into a form that is suitable for the later enumeration. The algorithm then starts with building a partial hypergraph by computing difference sets of *sampled* row pairs. For this partial hypergraph, the minimal hitting sets are enumerated using the *tree search* of `MMCS`. Whenever the search finds a new candidate solution, the *validation* checks whether the candidate is indeed a UCC. If so, it is a minimal UCC and `HPIValid` outputs it. If the validation shows that a column combination is not unique, we get an explicit list of yet indistinguishable records as a by-product. Every such pair gives a difference set that is not yet contained in the partial hypergraph. We resort again to sampling, focused on those pairs, to manage the influx of new sets. The tree search then resumes with the updated hypergraph from where it was paused.

4.1 Preprocessing

The only information from the database that is relevant for the enumeration of its UCCs is the grouping of rows with same value in every column. `HPIValid` uses a data structure

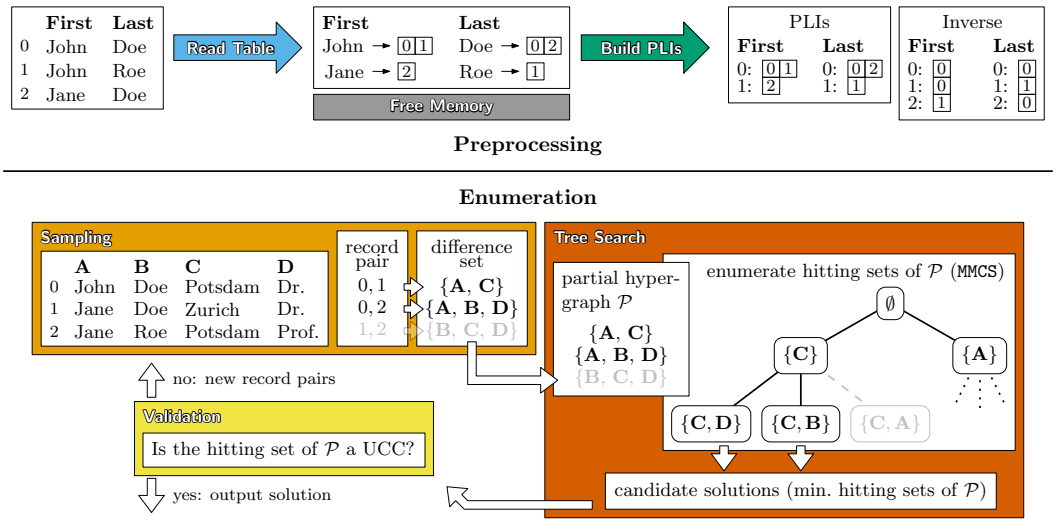


Figure 1: Overview of `HPIValid`. During preprocessing, the table is read and the preliminary cluster structure is extracted. The PLIs are created by copying and the memory is subsequently freed. Sampling generates a partial hypergraph of difference sets for the tree search to find new candidate solutions to validate.

called *position list indices*[1] (PLI) to hold this information, like many other data profiling algorithms [2, 12, 22, 23, 32, 33]. Each row is identified by some *record ID*. The PLI of an attribute a is an array of sets of record IDs. It has one set for each distinct value in a and the IDs in any set correspond to exactly those rows that have this value. The sets are called *clusters* and labeled by their *cluster ID*. We say a cluster is *trivial* if it has only a single row; they are not needed for the further computation. A PLI is thus a way to represent the *cluster structure*. They are not limited to single attributes and pertain also to combinations of columns.

The top part of Figure 1 shows an example. There are three records with IDs 0, 1, and 2. The values in attribute First are "John" for record 0 and 1 and "Jane" for record 2. The PLI of First thus contains two clusters. The cluster with ID 0 contains the record IDs 0 and 1. The cluster with ID 1 is trivial as it contains only the record ID 2.

The preprocessing consists of three steps. `HPIValid` first reads the table and, for each column, creates a preliminary data structure, a hash map, from the values to their respective clusters. In the second step, we extract the PLIs from the hash map. As same values do not necessarily appear in consecutive rows of the input file, the memory layout of the clusters is quite ragged after the initial read. To make the later enumeration more cache efficient, we create the actual PLIs by copying the data from the preliminary structure into consecutive memory. Besides the PLIs themselves, we also compute the inverse mappings from record IDs to cluster IDs [32], see Figure 1. This mapping is used in the validation of UCC candidates to efficiently intersect the PLI of a column combination with the PLI of a single attribute; hence, inverse mappings are needed only for single columns.

Finally, we free up the memory occupied by the preliminary data structure. Due to the ragged memory layout, this can take a material amount of time, which we report separately in the evaluation. This step could be avoided trading a higher memory footprint for a slightly smaller run time.

[1]Position list indices are also called *stripped partitions* [23].

4.2 Sampling

Sampling with respect to an attribute a means that we draw record pairs that coincide on a uniformly at random. In the language of PLIs, the PLI of a is a set of clusters $C_1, C_2, \ldots, C_n$ where each C_i contains records with equal values in a. Then there are $p = \sum_{i=1}^{n} \binom{|C_i|}{2}$ pairs that are indistinguishable with respect to a.[2] As p grows quadratically in the cluster size and thus in the number of rows, it is infeasible to compare all pairs by brute force. Instead, we fix some real number x between 0 and 1 as the *sampling exponent* and sample p^x record pairs. As long as there is sampling budget left, we select a cluster i with probability proportional to $\binom{|C_i|}{2}$ and then sample two rows from the cluster uniformly at random without replacement.

The sample exponent x enables control over the number of pairs and thus the time needed for the sampling. `HPIValid` allows the user to choose the sample exponent. Our experiments show that $x = 0.3$ is a robust choice, see Section 5.1.

Comparing the sampled pairs attribute-wise gives the partial hypergraph of difference sets. We need only its inclusion-wise minimal edges for the UCC discovery. To save space, we discard all non-minimal difference sets.

Initially, we sample once with respect to each attribute. As the resulting hypergraph $\mathcal{P}$ is potentially missing some edges, we have to resample later. This is done in such a way that every newly sampled record pair is guaranteed to yield an edge that is new to $\mathcal{P}$. We achieve this by letting the tree search (described in the next section) enumerate minimal hitting sets of $\mathcal{P}$. If the validation concludes that such a hitting set S of $\mathcal{P}$ is not actually a UCC, then there must be a so far unsampled record pair in the database that coincides on S. Thus, the PLI of S has non-trivial clusters and we sample with respect to S, i.e., we sample from all record pairs that are in the same cluster. Every difference found in this way is a witness for the fact that S is not a hitting set of the true hypergraph.

[2]Trivial clusters do not contribute to p since $\binom{1}{2} = 0$.

The example in Figure 1 has three records. Assume that the initial sampling returns only the record pairs $(0, 1)$ and $(0, 2)$ but not $(1, 2)$, i.e., the partial hypergraph $\mathcal{P}$ is missing an edge (grayed out). The tree search will later find the hitting set $S = \{A\}$. The validation concludes that S is not a UCC and resampling w.r.t. S yields the missing record pair. The tree search continues with an updated hypergraph.

Our sampling differs from the one of `HyUCC` in that we draw uniformly from a cluster without a pair-picking heuristic like pre-sorting or windowing. A heuristic would be less impactful on our algorithm, as we selectively choose only few record pairs from specific clusters, any time we sample.

4.3 Tree Search

As mentioned above, we use the `MMCS` algorithm [31] to enumerate minimal hitting sets of the current partial hypergraph. We can use it almost as a black box, except that we integrated the validation directly into the search tree to save computation time. We thus briefly discuss the information necessary to understand how the validation works.

The `MMCS` algorithm, working on the partial hypergraph $\mathcal{P}$, constructs a search tree of partial solutions. Every node of the tree maintains a set S of vertices (corresponding to attributes in the database) and the collection of those edges $E \in \mathcal{P}$ that are not yet hit by S, i.e., $E \cap S = \emptyset$. Based on a heuristic, `MMCS` then chooses an unhit edge E^*. As a hitting set must hit every edge, we in particular have to choose at least one vertex from E^*. `MMCS` now simply branches on the decision which vertex $v \in E^*$ to add to S. After branching, the search continues in a child node with the new set $S \cup \{v\}$. If there is no unhit edge left, we have found a hitting set.

In the example in Figure 1, the partial hypergraph $\mathcal{P}$ consists of two initially unhit edges. `MMCS` chooses the smaller one $\{A, C\}$ to branch on. It selects C in one branch and A in the other. After selecting C, the edge $\{A, B, D\}$ is still unhit and we branch on whether to include D, B, or A. The former two branches (with D and B) yield minimal hitting sets. For the last branch, `MMCS` recognizes that $\{C, A\}$ violates a minimality condition and prunes this branch.

4.4 Validation

As mentioned above, we adapt `MMCS` to directly integrate the validation. The tree search makes sure that we find the minimal hitting sets of the partial hypergraph. We additionally have to verify that this set is a UCC of the database $\mathfrak{r}$ (see Corollary 1). A set $S \cup \{v\}$ of attributes is a UCC if it partitions its subtuples $r[S \cup \{v\}]$, for all records $r \in \mathfrak{r}$, into only trivial clusters. We obtain the PLI for $S \cup \{v\}$ from the one for S and the single column v via PLI intersection [23] using the inverse PLIs for optimization [32]. Recall from the previous section that every node of the search tree corresponds to a set S of columns and that every child just adds a single column v to S. Thus, once we have the PLI for one node S in the search tree, the PLI intersect of S's and v's PLIs produces the PLI for any child $S \cup \{v\}$.

Suppose that we know the clusters $C_1, \ldots, C_n$ of set S, with trivial clusters already stripped. By intersecting the PLI of S with that of v, the C_i are subdivided into smaller groups corresponding to the same values in column v. For the single-column PLI of v we know the inverse mapping, so for each cluster ID i and every record ID identifying some $r[S] \in C_i$, we look up in which cluster of v this record lies. This gives us a new PLI now representing the cluster structure of $S \cup \{v\}$. Subdivided clusters that became trivial can again be stripped from the partition. Building these mappings scales with the total number of rows in the C_i.

If some set is found to be a hitting set of the partial hypergraph and its cluster structure is empty (contains only trivial clusters), we output it as a minimal UCC; otherwise, there are non-trivial clusters left and we sample new difference sets from these clusters as described in Section 4.2.

As some branches of the search tree do not produce hitting sets that have to be validated, it is not necessary to compute the PLIs for every node of the tree. Instead, we intersect the PLIs lazily only along branches that find a solution.

5. EVALUATION

We evaluate our algorithm `HPIValid` in four main aspects:

(1) **Parameter choice:** How should one choose the sample exponent x? Is there a single universally good choice, independent of the dataset? How robust is the algorithm against small changes in the parameter?
(2) **Performance:** How does `HPIValid` perform regarding run time and memory consumption? How does it compare to the stat-of-the-art solution `HyUCC`?
(3) **Scaling:** How does `HPIValid` scale with the number of rows and columns in the input?
(4) **Reasoning:** What makes `HPIValid` perform well? Which optimizations contribute to its performance?

Experimental setup. `HPIValid` is implemented in C++[3] and was compiled with GCC 10.1.0. All experiments were run on a GNU/Linux system with an Intel® Core™ i7-8700K 3.70 GHz CPU and 32 GB RAM main memory. Unless otherwise stated, we used a time limit of 1 h (3600 s). The comparison experiments with `HyUCC`, implemented in Java, were run with OpenJDK 13 and the heap memory limit set to 25 GB. For both, `HPIValid` and `HyUCC`, run times were measured by the algorithms themselves, excluding, e.g., the JVM startup time. Memory consumption was measured using the `time -f '%M'` command.

Comparability. Run time measurements always evaluate the specific implementation. The differences between `HPIValid` and `HyUCC` we observe with respect to total run time (Section 5.2) and scaling behavior (Section 5.3) are beyond what can be explained with implementation details or the difference between C++ and Java. We take the following additional measures to negate the differences between C++ and Java: We exclude datasets with very low execution times and use the `-server` flag to optimize Java code more aggressively at run time. We run the compiler and garbage collector in parallel with 4 threads each (`-XX+UseParallelGC -XX:ConcGCThreads=4 -XX:CICompilerCount=4`). Finally, we log the time the Java execution is suspended at safepoints and report the total execution time minus this suspension.

Run time breakdown. We regularly break down the total run time into the times required for specific subtasks performed by `HPIValid`. *Read table* denotes the time to read and parse the input, *build PLIs* refers to the construction of the data structures needed to store the clusters for each column as well as the inverse mapping. The preprocessing is completed by *freeing memory* that is no longer necessary for the enumeration. See Section 4.1 for details on the preprocessing phase. The enumeration phase consists of three tasks that

[3] hpi.de/friedrich/research/enumdat.html#HPIValid

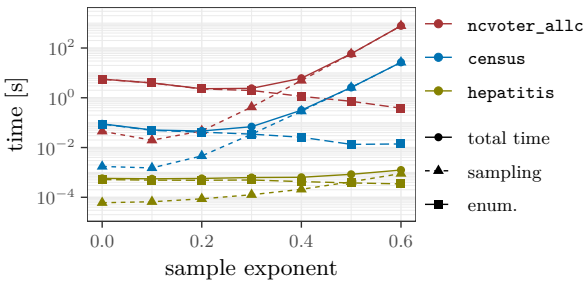


Figure 2: Run time scaling of `HPIValid` with respect to the sample exponent x in a log-plot. The time is shown without preprocessing, it is broken down into sampling time and the remaining enumeration. Data points correspond to medians of five runs.

do not occur in a fixed order but interleave. *Sampling* and *validation* respectively refer to the time spent to generate new difference sets (Section 4.2) and to validate candidate solutions (Section 4.4). Everything else is referred to as *tree search*, which is the time required by `MMCS` without the UCC-specific extensions. In the plots below, we associate specific colors with these subtasks, see Figure 3 for a legend. The same colors have already been used in Figure 1.

5.1 Parameter Choice

Recall that every time we encounter row pairs that are yet indistinguishable with respect to some candidate selection of columns, we compute the number p of such pairs and randomly sample p^x of them for an exponent $0 \leq x \leq 1$. Since p can be quadratic in the number of rows, it makes sense to choose $x \leq 0.5$. Larger values would yield a superlinear running time, which is prohibitively expensive.

In general, a smaller exponent x leads to fewer sampled row pairs, which should be beneficial for the run time. On the other hand, sampling fewer pairs leads to more inaccuracies in the partial hypergraph. It can be assumed that this lack of information misleads the algorithm in the enumeration phase, leading to higher run times. Our experiments confirm this intuition. Figure 2 shows run times on three datasets depending on x, divided into sampling time and enumeration time (tree search and validation). The general trend is that the sampling time increases with x, while the time for the remaining enumeration decreases.

In more detail, the sampling time resembles a straight line in the logarithmic plot for larger values of x. This corresponds to an exponential growth in x, which is to be expected since we sample p^x pairs. The constant or even slightly decreasing sampling time for very small values of x (e.g., between 0 and 0.1 for `census` and `ncvoter_allc` in Figure 2) is explained by the fact that we have to sample at least as many difference sets as there are edges in the correct hypergraph. Thus, if we sample fewer pairs, we have to sample more often. The enumeration time (excluding sampling) has a moderate downward trend. Also, for small values of x it is order of magnitudes higher than the sampling time and thus dominates the run time. The situation is reversed for larger values of x. There, the enumeration time decreases only slowly, while the sampling time goes up exponentially, making the sampling the dominant factor.

As a result, there is a large range between 0 and 0.4 where the total run time varies only slightly before the sampling time takes over. The minimum appears to be around 0.3. Preliminary experiments showed the same effect on other datasets. We thus conclude that $x = 0.3$ is a good choice for many datasets. Moreover, the fact that 0.3 lies in a wide valley of very similar run times makes it a robust choice. All other experiments in this paper were done with this setting.

5.2 Performance

To evaluate the performance of `HPIValid` and to compare it to `HyUCC`, we ran experiments on 62 datasets of different sizes from various domains. Additionally, we considered truncated variants for some datasets, mainly for consistency with related work [33], and to increase the comparability in cases where `HyUCC` exceeded the time limit. Table 1 shows the results sorted by run time, excluding thirteen datasets where `HPIValid` took less than 1 ms. For these instances, the speedup factor of `HPIValid` was between 74 and 640.

Run time performance. `HPIValid` solved all but two instances within the time limit of 1 h. The exceptions are `isolet`, and `uniprot` truncated at 1 k rows (but with the full 223 columns). Compared to the other instances, these datasets seem rather special as they have a huge number of UCCs. After 1 h, `HPIValid` enumerated for `isolet` and `uniprot` more than 153 M and 1743 M UCCs, which corresponds to 42 k and 484 k per second, respectively. To prevent I/O from obfuscating the actual run time, we do not output the UCCs but instead merely count the solutions.

Besides the two special cases, only `tpch_denormalized` came close to the time limit with its processing taking 2291 s. All other instances were solved in less than 3 min. Moreover, the breakdown of the run times in Table 1 into the different subtasks (see also Figure 3) shows that preprocessing usually makes up more than half of the total run time. The overall performance therefore cannot be significantly improved without improving the preprocessing. There are some interesting exceptions to this trend: On the instances `lineitem`, `ncvoter_allc` and `tpch_denormalized`, `HPIValid` spent the majority of the execution on the validation. We discuss this effect further in the scaling experiments in Section 5.3. For the mentioned instances `isolet` and `uniprot` truncated at 1 k rows, the algorithm spent by far the most time with the tree search. The same holds true for these datasets respectively truncated at 200 and 120 columns. Improving upon these run times requires to improve `MMCS`, the state-of-the-art in enumerating minimal hitting sets [19, 31].

Memory consumption. Concerning the consumption of main memory, `HPIValid` tops out at 13 GB, with only three of the datasets requiring more than 10 GB. For those three, already the input size is rather large: the `ncvoter_allc` dataset has 7.5 M rows, `VTTS` has 13 M, and `iloa` 20 M rows.

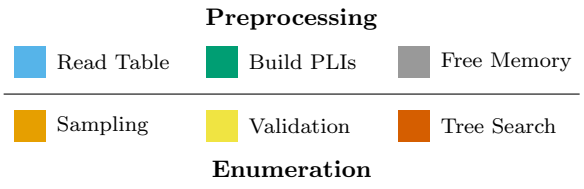
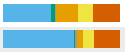
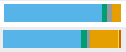
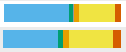
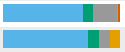
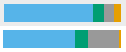
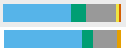
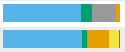
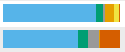
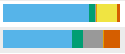
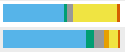


Figure 3: Color-coding of the run time breakdown.

Table 1: Run times and memory consumption of HPIValid and HyUCC. All times shown are the median of five runs, except when the algorithm hit the timeout, in which case it is only a single run. The run times for HPIValid are broken down according to the color-coding in Figure 3. The empty entries for HyUCC indicate that the 25 GB heap memory did not suffice. For HyUCC, the reported times exclude the time the JVM suspended execution at safepoints. The actual execution times including this are given in the 'Total' column. [a]'UCCs' column shows the number of solutions HPIValid enumerated within the time limit of 3,600 s. [b]Subsets of another dataset appearing in the table. [c]Shrunk from 45 M rows to 20 M to fit into memory. [d]Single run with a time limit of 28,800 s for HyUCC.

Dataset	Rows [#]	Cols [#]	UCCs [#]	HPIValid Time [s]	HPIValid Time Breakdown	HPIValid Mem [MB]	HyUCC Comparison Total [s]	HyUCC Comparison Time [s]	HyUCC Comparison Mem [MB]	Speedup
horse	300	29	253	1.14 m		8	0.10	0.10	76	92.27
amalgam1_denormalized	51	87	2,737	1.67 m		8	0.07	0.07	77	44.97
t_bioc_measurementsorfacts	3.11 k	24	2	3.79 m		9	0.09	0.09	70	22.41
plista	996	63	1	4.67 m		9	0.29	0.29	83	61.39
nursery	13.0 k	9	1	5.74 m		10	0.14	0.14	82	23.54
t_bioc_specimenunit_mark	8.98 k	12	2	8.48 m		11	0.14	0.14	87	16.62
chess	28.1 k	7	1	9.43 m		12	0.15	0.15	100	15.90
letter	18.7 k	17	1	0.02		12	0.43	0.43	132	22.53
flight	1.00 k	109	26,652	0.03		10	1.50	1.48	622	56.02
t_bioc_multimediaobject	18.8 k	15	4	0.04		27	0.28	0.28	133	7.28
SG_TAXON_NAME	106 k	3	2	0.05		26	0.27	0.27	173	5.57
entytysrcgen	26.1 k	46	3	0.08		31	1.33	1.33	207	17.41
t_bioc_gath_agent	72.7 k	18	4	0.11		41	0.51	0.50	230	4.46
Hospital	115 k	15	12	0.12		48	0.88	0.87	231	7.52
t_bioc_preparation	81.8 k	21	2	0.12		46	0.51	0.50	229	4.19
SPStock	122 k	7	14	0.14		37	0.56	0.55	226	3.93
t_bioc_gath_sitecoordinates	91.3 k	25	2	0.17		56	0.62	0.61	245	3.53
t_bioc_gath_namedareas	138 k	11	4	0.18		55	0.85	0.83	235	4.62
t_bioc_ident_highertaxon	563 k	3	1	0.18		56	1.08	1.06	374	5.71
t_bioc_gath	91.0 k	35	1	0.19		59	1.02	1.00	350	5.36
t_bioc_unit	91.3 k	14	2	0.22		63	0.61	0.59	253	2.66
SG_BIOENTRY_REF_ASSOC	358 k	5	1	0.25		66	0.86	0.84	294	3.34
t_bioc_ident	91.8 k	38	2	0.34		82	1.21	1.18	400	3.48
musicbrainz_denormalized	79.6 k	100	2,288	0.45		125	22.24	22.19	1,155	49.22
census	196 k	42	80	0.48		160	394.57	385.19	3,972	800.69
SG_BIOSEQUENCE	184 k	6	1	0.49		143	1.06	1.01	486	2.07
SG_REFERENCE	129 k	6	3	0.51		113	0.67	0.62	341	1.20
SG_BIOENTRY	184 k	9	3	0.52		106	0.78	0.74	308	1.42
SG_BIOENTRY_QUALIFIER_ASSOC	1.82 M	4	2	0.72		201	3.70	3.62	798	5.02
SG_SEQFEATURE_QUALIFIER_ASSOC	825 k	4	1	0.76		156	1.25	1.18	486	1.55
SG_BIOENTRY_DBXREF_ASSOC	1.85 M	3	2	0.79		193	2.43	2.37	701	2.98
SG_DBXREF	618 k	4	2	0.89		158	0.88	0.80	513	0.89
SG_SEQFEATURE	1.02 M	6	2	0.97		196	2.19	1.98	894	2.05
ncvoter_allc[b]	100 k	94	15,244	1.02		193	184.42	184.06	5,920	180.79
Tax	1.00 M	15	13	1.46		214	9.70	9.63	1,352	6.58
SG_LOCATION	1.02 M	8	2	1.76		305	2.26	2.07	1,014	1.17
uniprot[b]	1.00 k	120	1,973,734	1.93		12	32.47	31.38	8,229	16.24
struct_sheet_range	664 k	32	167	4.14		623	17.89	17.24	2,277	4.17
fd-reduced-30	250 k	30	3,564	4.36		187	115.15	114.89	2,297	26.34
CE4HI01	1.68 M	65	25	6.88		1,514	34.29	33.48	4,674	4.87
ZBC00DT_COCM	3.18 M	35	1	9.60		1,752	34.43	32.57	5,233	3.39
isolet[b]	7.80 k	200	1,282,903	11.12		132	249.74	249.04	2,267	22.40
ditag_feature	3.96 M	13	3	13.09		1,736	109.86	106.41	4,637	8.13
ncvoter_allc[b], [d]	1.50 M	94	206,220	17.12		2,626	>28,800	>28,771	13,701	>1,680
uniprot	539 k	223	826	19.56		2,746				
PDBX_POLY_SEQ_SCHEME	17.3 M	13	5	24.40		4,394	83.53	73.52	11,705	3.01
ncvoter	8.06 M	19	96	47.50		4,226	286.48	274.61	10,471	5.78
ILOA[c]	20.0 M	48	1	48.26		13,208	235.67	217.32	21,599	4.50
VTTS	13.0 M	75	2	60.48		12,373	384.00	373.25	19,098	6.17
lineitem	6.00 M	16	390	81.34		2,234	454.29	448.56	7,172	5.51
ncvoter_allc	7.50 M	94	1,704,511	124.77		12,471	>3,600	>3,549	20,742	>28
tpch_denormalized	6.00 M	52	347,805	2,291.33		9,331	>3,600	>3,586	13,623	>1
uniprot[a], [b]	1.00 k	223	>1,743 M	>3,600		13				
isolet[a]	7.80 k	618	>153 M	>3,600		295				

Nine datasets required between 1 GB and 10 GB, and all remaining datasets took less memory.

Memory consumption compared to HyUCC. When comparing HPIValid and HyUCC, one striking difference is the fact that HyUCC has to keep its search front of column combinations in memory. In particular, the front includes all minimal UCCs found so far, while HPIValid only needs to store the current branch of the search. It has thus a significantly larger memory footprint, especially on instances with many solutions. In fact, this makes it infeasible to process the two extreme datasets isolet and uniprot truncated at 1 k rows with HyUCC. The variants in which the number of columns are cut at 200 for isolet and 120 for uniprot can still be solved. However, on the former, HPIValid is more memory efficient than HyUCC by an order of magnitude. On uniprot with 200 columns and 1 k rows, it is by almost three orders of magnitude: HyUCC requires 8 GB, HPIValid 12 MB.

It is curious to see that the full uniprot dataset with 539 k rows is much more well-behaved than the one truncated at 1 k rows. One could expect that larger databases lead to higher run times, and indeed this is the case for most databases, see Section 5.3. However, the uniprot dataset is special in that the extended instance has only 826 UCCs and HPIValid can solve it in under 20 s using 2.7 GB. Recent theoretical work on random hypergraphs shows that a larger number of rows and more difference sets can indeed result in hypergraphs with fewer and smaller edges [8]. HyUCC on the other hand cannot solve the full uniprot instance due to memory overflow. Thus, the fact that uniprot includes a hard subinstance seems to throw off HyUCC even though the full instance contains only few UCCs. On all remaining instances, HPIValid is also more memory efficient.

Run time compared to HyUCC. HPIValid outperformed HyUCC on every instance with only one exception. On some instances with very few UCCs, HyUCC achieves comparable run times. On many other instances HPIValid was significantly faster than HyUCC. The highest speedup achieved on instances that HyUCC finished was for the census dataset, HPIValid was 800 times faster. On the ncvoter_allc dataset truncated at 1.5 M rows, we ran it with an 8 h timeout, which was exceeded by HyUCC, while HPIValid solved it in below 20 s, a speedup of at least three orders of magnitude.

5.3 Scaling

We now evaluate how HPIValid scales with respect to the number of rows and columns. To provide some context, we preface our experiments with a short discussion on worst-case run times from a slightly more theoretical perspective.

Regarding the column scaling, the bottleneck is the actual enumeration (tree search and validation). As there can be exponentially many minimal hitting sets and as many UCCs, the worst-case running time must be exponential [6]. However, even if the number of solutions is small, there is no subexponential upper bound known for the MMCS algorithm, which is the core of HPIValid. We thus have to assume that HPIValid scales exponentially with the number of columns, even if the output is small. It is a major open question whether an output-polynomial algorithm exists for the hitting set enumeration problem, see [15].

Concerning the number of rows, in principle, we have to compute the difference set of every record pair, which scales quadratically in the number of rows. Moreover, when building the hypergraph of difference sets, we only keep edges that

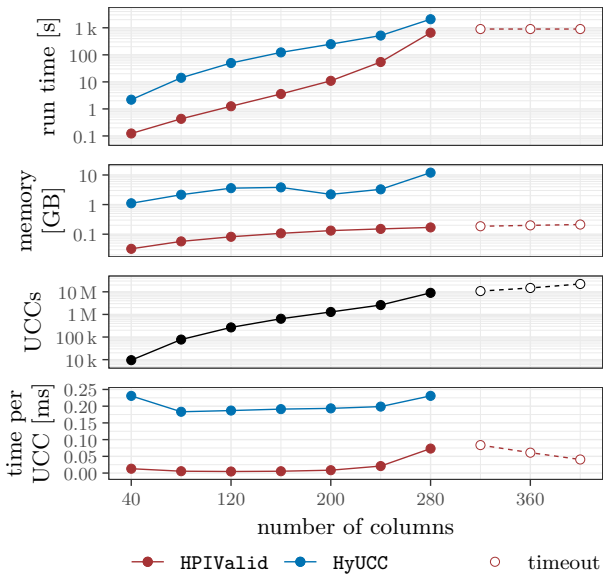


Figure 4: Column scaling of HPIValid and HyUCC on isolet. Shown are the run times, the memory consumption, the number of solutions (log-plots), and the average delay between outputs. Data points correspond to the median of three runs. Dashed lines show runs of HPIValid with 15 min timeout.

are minimal, without affecting the solutions. For each new edge we sample, this takes time linear in the number of edges currently in the hypergraph. This means a quadratic run time in the number of difference sets. Thus, the best worst-case upper bound for the run time in terms of the number of rows n is $O(n^4)$. Moreover, there are lower bounds known for the minimization step based on the strong exponential time hypothesis (SETH), implying that there is likely no algorithm with subquadratic running time in n [11, 20].

Although these worst-case bounds seem to prohibit the solution of large instances, HPIValid performs well on practical datasets. The reason for this lies in the fact that these databases behave very differently from worst-case instances. Real-world instances have only comparatively few minimal difference sets [9, 10] and indeed HPIValid finds them by a focuses sampling approach involving only a few record pairs. Similarly, the tree search algorithm MMCS has been observed to be fast on hypergraphs arising in practice [19, 31], and the instances emerging from the UCC enumeration problem are no exception to this. The only outliers in our experiments in that regard are isolet and uniprot truncated at 1 k rows, where the hitting set enumeration is slow, which is not surprising due to their large output size.

Consequently, the goal of our experiments is the gathering of insights into the behavior of HPIValid on *typical* datasets rather than worst-case instances. The emphasis of our scaling experiments is on databases other than isolet. Nonetheless, we have a short section discussing the column scaling for this as well since isolet has by far the most columns among all the tested datasets, and it has been considered before for scaling experiments [33]. Although experiments on isolet may not reflect the typical scaling, we can still make some interesting observations, in particular on the differences between HPIValid and HyUCC.

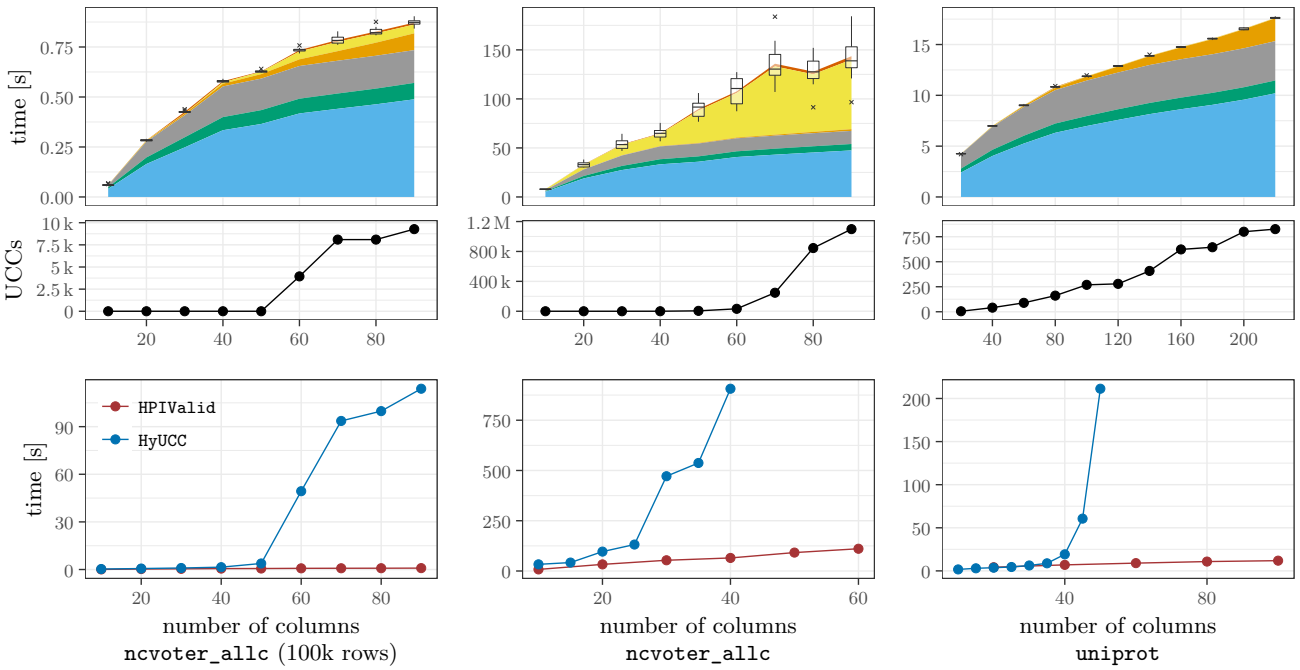


Figure 5: Run time scaling of `HPIValid` with respect to the number of columns. Each column shows one dataset. The top row shows the run time averaged over 15 runs of `HPIValid` broken down according to the color-coding in Figure 3. The middle row shows the number of UCCs. The bottom row compares the median run times of `HPIValid` with that of `HyUCC` with five runs each. Note the different abscissas in the bottom row.

5.3.1 *Column Scaling on the* `isolet` *dataset*

We used the first 40 to 280 columns of `isolet`, stride 40, and ran `HPIValid` and `HyUCC` on the resulting instances. Since the run times for `isolet` are fairly concentrated over the different runs, we used the median of three. We also ran `HPIValid` with a timeout of 15 min for number of columns beyond 280. The results are shown in Figure 4.

The run time of `HPIValid` resembles a straight line in the plot with a logarithmic time-axis, meaning an exponential scaling. As discussed before, this is to be expected on certain classes of inputs. In contrast, it is interesting to see that the run time of `HyUCC` as well as the number of UCCs (second plot) seems to scale subexponentially. A possible explanation is that `HPIValid` uses the branching technique of `MMCS`, which potentially scales exponentially, regardless of the output. `HyUCC` on the other hand explores the search space of all column combinations starting with the smaller subsets. Given that the solutions for `isolet` are essentially all column combinations of size 3, then `HyUCC`'s run time is dominated by the output size, which grows cubically here.

Another indicator that the run time of `HPIValid` scales worse than the number of UCCs is the uptick for the time per UCC in the bottom plot of Figure 4. The last experiments with timeout effectively provide a snapshot of the first 15 min of execution with many columns. There the trend in the time per UCC is reversed. Although the average delay over the entire run goes up, it is decreasing at the start. This is helpful if one is interested in getting only a few UCCs.

5.3.2 *Column Scaling on Typical Instances*

We chose two datasets to investigate typical column scaling. The `uniprot` dataset (with all rows) is an obvious choice with its 223 columns. The other dataset, `ncvoter_allc`, is one of the hardest instances for `HPIValid` with a run time of 125 s, and it has enough columns (94) to enable meaningful scaling experiments. For additional comparison with `HyUCC`, we also ran experiments on `ncvoter_allc` truncated at 100 k rows. The results are shown in Figure 5. We discuss them instance by instance from left to right. For the color-coding of the run time breakdown, recall Figure 3.

Truncating `ncvoter_allc` at 100 k rows makes it small enough so that the preprocessing dominates the run time of `HPIValid`. It stands out that the run time appears to scale sublinearly with respect to the number of columns. In theory, this cannot happen for an algorithm that processes the whole input at least once. The reason for the observed behavior are the later columns in the table having fewer different values and more empty cells. This makes reading the table faster as the hash map matching string values to arrays of record IDs needs to cover a smaller domain during dictionary encoding in the preprocessing.

The actual enumeration times are very low with a slight uptick after 50 columns. This is due to the fact that the nature of the instance changes markedly here. The output size increases from a single minimal UCC for the first 50 columns to 4 k solutions at 60 columns. The further increase to 8 k minimal UCCs for 70 columns is not reflected in the enumeration time, starting at 60 columns. Instead, the enumeration time scales linearly in this experiment. The bottom plot shows the scaling behavior of `HyUCC` in comparison. `HyUCC` also performs well on instances with a single solution. However, the scaling of `HyUCC` beyond that point (60+ columns) does not seem to be linear in the number of columns but rather follows the number of UCCs.

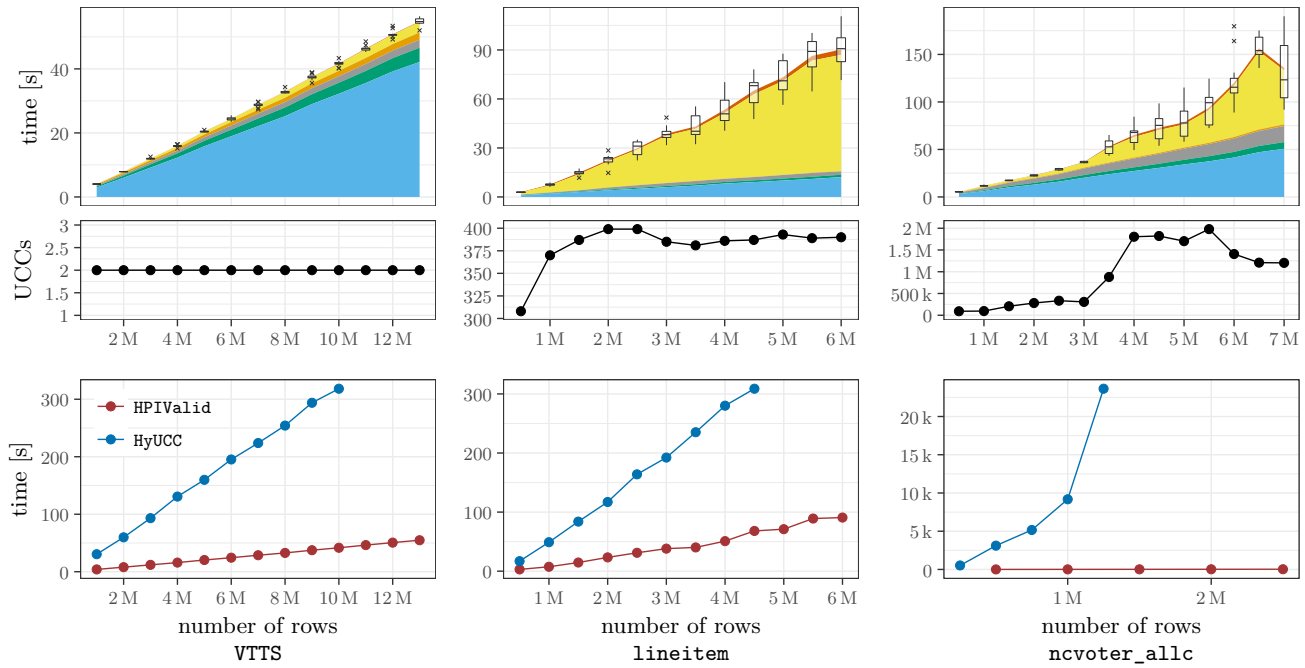


Figure 6: Run time scaling of `HPIValid` with respect to rows. Each column represents one dataset. The top row averages over 15 runs of `HPIValid` broken down according to the colors in Figure 3. The middle shows the number of UCCs. The bottom row compares the median run times of `HPIValid` with that of `HyUCC` with five runs each for `VTTS` and `lineitem`. Only one `HyUCC` run was done for `ncvoter_allc`, note the different abscissa.

The middle column of Figure 5 shows the `ncvoter_allc` dataset with all 7.5 M rows. The output size is also small in the beginning, but there is always more than one solution. The overall column scaling is roughly linear. However, the box plots show that the run time has a high variance, which mainly comes from the validation. A possible explanation is as follows. Recall that every node in the `MMCS` search tree corresponds to a set of columns that was selected to be part of the solution. The core operation of the validation is the intersection of the PLI of this subset with the PLI of a single column. The time required for the intersection is linear in the number of rows that are not already unique with respect to the selected columns. If the search happens to select columns high up in the tree that make many rows unique, all intersections in the lower subtrees are sped up. Whether this occurs in any given run of the algorithm heavily depends on the difference sets present in the partial hypergraph when starting the enumeration. As the initial partial hypergraph is random to some extent, the run times vary strongly. The run time breakdowns in the top row of Figure 5 show average values and thus give an estimation of the expected running time of our randomized algorithm (in addition to the median values of the runs shown in the box plots).

The comparison with `HyUCC` on `ncvoter_allc` with full rows is difficult due to the large total run times. The measurements for `HyUCC` in the bottom plot are thus restricted to the instances with up to 40 columns. The scaling of `HyUCC` is clearly worse, although there are only few UCCs in the considered range (403 UCCs at 40 columns).

Finally, the right column of Figure 5 shows that the run time of `HPIValid` on `uniprot` is again dominated by the preprocessing. The enumeration part in turn consists mainly of sampling new difference sets rather than the validation, as was the case for `ncvoter_allc`. This makes sense in the light of the results in Section 5.2. Truncating the dataset at 1 k rows gives a hard subinstance with billions of minimal solutions. It thus cannot suffice to sample only a few record pairs and hope to come close to the correct hypergraph. In other words, redundant information seems to be not as prevalent in `uniprot` as in other databases.

The bottom plot shows that the scaling of `HyUCC` is super-linear in the number of columns. As observed before, `HyUCC` appears to have difficulties with `uniprot` due its hard subinstance. In Section 5.2, this became apparent with respect to the memory consumption. Here, it also pertains to the run time. Thus, even with hypothetically unbounded memory, the run time of `HyUCC` is infeasibly high.

5.3.3 *Row Scaling on Typical Instances*

For the row scaling experiments, we chose the three datasets `VTTS`, `lineitem`, and `ncvoter_allc`. There `HPIValid` had the highest run time (apart from `tpch_denormalized`). They have 13 M, 6 M, and 7.5 M rows, respectively, which makes them well suited for scaling experiments with respect to the number of rows. The results are shown in Figure 6. We note that many aspects discussed in Section 5.3.2 apply here as well, we thus focus on the specifics of the row scaling.

`VTTS` is the largest of the selected datasets, but `HPIValid` has the lowest run time there. The database carries only two minimal UCCs independently of the number of rows, making the linear preprocessing dominant.

The number of solutions for `lineitem` is also rather steady but on a higher level. It ranges from 300 to 400 suggesting that also the correct hypergraphs of difference sets vary

not too much for the different numbers of rows, given that a critical minimum value is exceeded. This gives a clean straight line for the average run time and most of this time is spent on the validation. For reasons already discussed in Section 5.3.2, the validation time has a comparatively high variance as indicated by the box plots.

In stark contrast to the other two datasets, the output size varies heavily for `ncvoter_allc`. With 500 k to 3 M rows, the number of solutions lies at between 91 k to 335 k, with 3 M to around 4 M rows, there is a jump to 1.8 M minimal UCCs. After this, the output size remains high until it goes down to 1.2 M at 7 M rows. Thus, the underlying hypergraph of minimal difference sets also changes significantly when adding more and more rows. This is reflected in the trend of the average run time being not as clean as the one for `lineitem`, although for both datasets the execution of `HPIValid` is dominated by the validation. Additionally, the run time has high variance again, the effect is even more prominent here due to the higher number of columns (`lineitem` has 16, while there are 94 in `ncvoter_allc`).

When comparing the row scaling of `HPIValid` with that of `HyUCC`, `VTTS` and `lineitem` are quite similar. Both algorithms seem to scale linearly, but with `HyUCC` having a steeper slope. For `ncvoter_allc`, the scaling is very different (note the different abscissa). `HPIValid` scales slightly superlinear, while `HyUCC` shows a sudden jump on instances truncated at more than 1 M. Further scaling experiments became infeasible even with an extended time limit of 8 h

5.4 Reasons for Efficiency

There are two crucial factors for the efficiency of `HPIValid`. First, the number of record pairs for which we compute difference sets, and secondly, the size of the search tree.

Concerning the number of difference sets, we initially sample $p^{0.3}$ record pairs where p is the total number of available pairs. This is sublinear in the number of records and thus always fast. However, the resulting hypergraph can be incomplete, which forces us to resample additional pairs later on. We measure this using the *relative resample rate*, which is the number of difference sets computed after the initial sampling divided by the number of difference sets computed in the initial sampling. Excluding two outliers, all instances considered in Section 5.2 have a relative resample rate below 0.36, with a median of 0.00038 and a mean of 0.041. The two outliers are the somewhat special `isolet` instances with different column numbers. They had relative resampling rates of 15.1 and 10.0, respectively.

Concerning the search tree, the number of leaves is at least the number of solutions. In the best case, the tree consists of just the root together with one child per solution. To measure the efficiency of the tree search, we consider the *solution overhead*, i.e., the number of non-root nodes per solution. For all instances with at least two solutions, the maximum overhead was 4.5, with a median of 1.4 and mean 1.7. This shows that the `MMCS` tree search enumerates hitting sets very efficiently even when working with partial information. On instances with only a single solution the overhead equals the total tree size. We got a maximum of 54, with a median of 5.0 and mean 10.5. Although these numbers are higher, they still indicate small search trees.

The efficiency of `HPIValid` is mainly due to the aspects discussed above. Beyond that, we have two minor optimizations that slightly improve the run time. First, to make

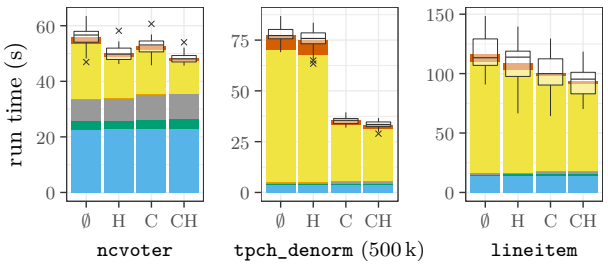


Figure 7: Run times with and without tiebreaking heuristic (H) and PLI copying (C) for the datasets `ncvoter`, `tpch_denormalized` (reduced to 500 k rows), and `lineitem`. Each column is based on 25 runs.

the validation more cache efficient, we copy the PLIs such that each PLI lies in a consecutive memory block. Secondly, whenever `MMCS` has multiple edges of minimum cardinality to branch on, we use a tiebreaker that aims at speeding up the validation by making the clusters in the resulting PLIs small. For this, we rank the columns by *uniqueness*, with a lower number of indistinguishable record pairs with respect to that column meaning higher uniqueness. Among the edges with minimum cardinality, we choose the one where the least unique column is most unique.

On instances where the `HPIValid` run time is dominated by the preprocessing, the effect of these two optimizations is negligible. However, one can see in Figure 7 that both optimizations improve the validation time at the cost that the PLI copying slightly increasing the preprocessing time.

6. CONCLUSION

We proposed a novel approach for the discovery of unique column combinations. It is based on new insights into the hitting set enumeration problem with partial information, where the lack of relevant edges is compensated by a validation procedure. Our evaluation showed that our algorithm `HPIValid` outperforms the current state of the art. On most instances, the enumeration times are so small that they are dominated by the preprocessing. This indicates that the room for further improvements is somewhat limited. We believe that it is much more promising to study how our new techniques can be used to solve other problems. Embedded uniqueness constraints, for example, are a generalization of UCCs to incomplete datasets with a similar discovery process [35]. Also closely related are functional dependencies: one can transform their discovery into a hitting set problem as well, only with slightly different hypergraphs. As for UCCs, the direct translation is infeasible in practice due to the quadratic number of record pairs, but it seems that recently proposed algorithms [32, 36] could be accelerated by enumeration with partial information. Similarly, we believe that our approach can work for denial constraints [9, 27].

Acknowledgements. The authors would like to thank Takeaki Uno and the attendees of the 3rd International Workshop on Enumeration Problems and Applications (WEPA 2019) for the fruitful discussions on hitting set enumeration.

7. REFERENCES

[1] Z. Abedjan, L. Golab, F. Naumann, and T. Papenbrock. *Data Profiling*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, San Rafael, CA, USA, 2018.

[2] 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.

[3] Z. Abedjan, J. Quiané-Ruiz, and F. Naumann. Detecting Unique Column Combinations on Dynamic Data. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1036–1047, 2014.

[4] P. Atzeni and N. M. Morfuni. Functional Dependencies and Constraints on Null Values in Database Relations. *Information and Control*, 70(1):1–31, 1986.

[5] C. Beeri, M. Dowd, R. Fagin, and R. Statman. On the Structure of Armstrong Relations for Functional Dependencies. *Journal of the ACM*, 31(1):30–46, 1984.

[6] J. C. Bioch and T. Ibaraki. Complexity of Identification and Dualization of Positive Boolean Functions. *Information and Computation*, 123(1):50–63, 1995.

[7] T. Bläsius, T. Friedrich, and M. Schirneck. The Parameterized Complexity of Dependency Detection in Relational Databases. In *Proceedings of the International Symposium on Parameterized and Exact Computation (IPEC)*, pages 6:1–6:13, 2016.

[8] T. Bläsius, T. Friedrich, and M. Schirneck. The Minimization of Random Hypergraphs. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 80:1–80:15, 2020.

[9] T. Bleifuß, S. Kruse, and F. Naumann. Efficient Denial Constraint Discovery with Hydra. *PVLDB*, 11(3):311–323, 2017.

[10] T. Bläsius, T. Friedrich, J. Lischeid, K. Meeks, and M. Schirneck. Efficiently Enumerating Hitting Sets of Hypergraphs Arising in Data Profiling. In *Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*, pages 130–143, 2019.

[11] M. Borassi, P. Crescenzi, and M. Habib. Into the Square: On the Complexity of Some Quadratic-time Solvable Problems. *Electronic Notes in Theoretical Computer Science*, 322:51–67, 2016.

[12] S. S. Cosmadakis, P. C. Kanellakis, and N. Spyratos. Partition Semantics for Relations. *Journal of Computer and System Sciences*, 33(2):203–233, 1986.

[13] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley Longman Publishing, Boston, MA, USA, 8th edition, 2003.

[14] J. Demetrovics and V. D. Thi. Keys, Antikeys and Prime Attributes. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae Sectio Computatorica*, 8:35–52, 1987.

[15] T. Eiter, K. Makino, and G. Gottlob. Computational Aspects of Monotone Dualization: A Brief Survey. *Discrete Applied Mathematics*, 156(11):2035–2049, 2008.

[16] R. Fadous and J. Forsyth. Finding Candidate Keys for Relational Data Bases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 203–210, 1975.

[17] F. V. Fomin, F. Grandoni, A. V. Pyatkin, and A. A. Stepanov. Combinatorial Bounds via Measure and Conquer: Bounding Minimal Dominating Sets and Applications. *ACM Transactions on Algorithms*, 5(1):9:1–9:17, 2008.

[18] V. Froese, R. van Bevern, R. Niedermeier, and M. Sorge. Exploiting Hidden Structure in Selecting Dimensions That Distinguish Vectors. *Journal of Computer and System Sciences*, 82(3):521–535, 2016.

[19] A. Gainer-Dewar and P. Vera-Licona. The Minimal Hitting Set Generation Problem: Algorithms and Computation. *Journal on Discrete Mathematics*, 31(1):63–100, 2017.

[20] J. Gao, R. Impagliazzo, A. Kolokolova, and R. Williams. Completeness for First-order Properties on Sparse Structures with Algorithmic Applications. *ACM Transactions on Algorithms*, 15(2):23:1–23:35, 2018.

[21] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[22] A. Heise, J.-A. Quiané-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann. Scalable Discovery of Unique Column Combinations. *PVLDB*, 7(4):301–312, 2013.

[23] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *The Computer Journal*, 42(2):100–111, 1999.

[24] H. Köhler, U. Leck, S. Link, and X. Zhou. Possible and Certain Keys for SQL. *VLDB Journal*, 25:571–596, 2016.

[25] H. Köhler, S. Link, and X. Zhou. Discovering Meaningful Certain Keys from Incomplete and Inconsistent Relations. *IEEE Data Engineering Bulletin*, 39(2):21–37, 2016.

[26] V. B. T. Le, S. Link, and F. Ferrarotti. Empirical Evidence for the Usefulness of Armstrong Tables in the Acquisition of Semantically Meaningful SQL Constraints. *Data & Knowledge Engineering*, 98:74–103, 2015.

[27] E. Livshits, A. Heidari, I. F. Ilyas, and B. Kimelfeld. Approximate Denial Constraints. *PVLDB*, 13(10):1682–1695, 2020.

[28] C. Mancas. Algorithms for Database Keys Discovery Assistance. In *Proceedings of the International Conference on Perspectives in Business Informatics Research (BIR)*, pages 322–338, 2016.

[29] H. Mannila and K.-J. Räihä. Dependency Inference. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 155–158, 1987.

[30] H. Mannila and K.-J. Räihä. Algorithms for Inferring Functional Dependencies from Relations. *Data & Knowledge Engineering*, 12(1):83–99, 1994.

[31] K. Murakami and T. Uno. Efficient Algorithms for Dualizing Large-scale Hypergraphs. *Discrete Applied Mathematics*, 170:83–94, 2014.

[32] T. Papenbrock and F. Naumann. A Hybrid Approach to Functional Dependency Discovery. In *Proceedings*

of the International Conference on Management of Data (SIGMOD), pages 821–833, 2016.

[33] T. Papenbrock and F. Naumann. A Hybrid Approach for Efficient Unique Column Combination Discovery. In *Proceedings of the Conference Datenbanksysteme in Business, Technologie und Web Technik (BTW)*, pages 195–204, 2017.

[34] 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 Data Bases (VLDB)*, pages 691–702, 2006.

[35] Z. Wei, U. Leck, and S. Link. Discovery and Ranking of Embedded Uniqueness Constraints. *PVLDB*, 12(13):2339–2352, 2019.

[36] Z. Wei and S. Link. Discovery and Ranking of Functional Dependencies. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1526–1537, 2019.